Materialsammlung - Formale Methoden: OCL und Eclipse

Prof. Dr. Hans-Jürgen Buhl

Sommersemester 2009
Fachgruppe Mathematik und Informatik
FB C
Bergische Universität Wuppertal

Praktische Informatik
PIBUW - WS09/10
Oktober 2009
4. Auflage, 2009
Praktische Informatik 02

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

092MAT512000

Formale Methoden
4 V Di 12 - 14 D.13.08
Do 12 - 14 D.13.08
Einordnung: Master IT; Master Mathematik; Nebenfächer und Studienschwerpunkte Informatik anderer Studiengänge

Inhalt: Softwarequalität, Zusicherungen, Klassifizierung von Klassenmethoden; Programming by Contract; Vorbedingungen, Nachbedingungen und Invarianten; Contracts bei der Vererbung; formale Spezifikation mit OCL2 und Eclipse; Frame-Regeln; Fallstudien formaler Spezifikation.

092MAT512001

Übungen zu Formale Methoden
2 Ü Mi 16 - 18 D.13.08

Vorbemerkungen:

Formale Methoden

Inhalte:

1. Softwaregüte

2. Zusicherungen in Algorithmen:
   Konstruktoren, Modifikatoren,
   Observatoren und Destruktoren;
   Ausnahmebedingungen

3. Methodik *Programming by Contract*:
   Vorbedingungen, Nachbedingungen und Invarianten;
   Softwareanbieter/Softwarenutzer

4. Startwerte, Vererbung von Klasseninvarianten,
   Methodenvor- und -nachbedingungen

5. Formale Spezifikation (in OCL2):
   UML-Klassendiagramme und *Constraints*
   virtuelle Attribute und Methoden,
   redundante Attribte und Methoden;
   Constraints an Attribte, Methoden und
   Assoziationen; Container-Typen; Frame-Regeln

6. Fallstudien von formal spezifizierter
   Software (Algorithmen und Datenstrukturen)

7. Von der formalen Spezifikation zur (Prototyp-)Software

Modulziele:

Die Studierenden lernen formale Software-Modelle lesen, verstehen und kritisieren, um formale Methoden als ein Kommunikationsmittel der Teammitglieder eines Software-Entwicklungsteams schätzen zu lernen. Sie entwickeln mit Hilfe der formalen Spezifikation Teilsysteme von realistischen Softwaremodellen selbst.

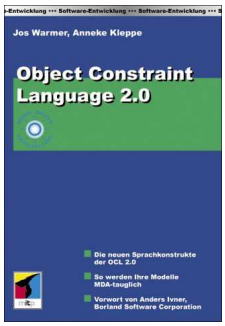
Literatur:

Wolfgang Zuser
Software Engineering
Mit UML und dem Unified Process
Gebundene Ausgabe - 464 Seiten
Pearson Studium
Erscheinungsdatum: Juni 2004

Auflage: 2., überarb. Aufl.
ISBN: 3827370906

Jos Warmer
Object Constraint Language2.0
Broschiert - 240 Seiten
Mitp-Verlag
Erscheinungsdatum: März 2004

ISBN: 3826614453



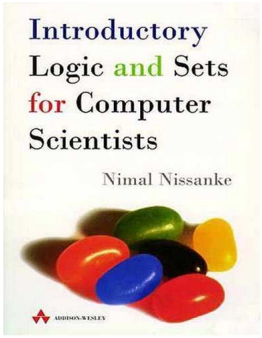
OMG
Object Constraint Language
OMG Available Specification
Versionb 2.0
http://www.omg.org/docs/formal/06-05-01.pdf

Tony Clark, Jos Warmer
Object Modeling with the OCL.
The Rationale behind the Object Constraint Language
http://www.amazon.de/Object-Modeling-OCL-Rationale-Constraint/dp/3540431691
ISBN: 3-540-43169-1

Nimal Nissanke
Introductory Logic and Sets for Computer Scientists.

Broschiert - 400 Seiten
Addison Wesley
Erscheinungsdatum: Oktober 1998

ISBN: 0201179571



Martin Kreuzer, Stefan Kühling
Logik für Informatiker

Pearson Studium
Erscheinungsdatum: März 2006

ISBN: 3827372151

Dan Pilone
UML 2.0 kurz & gut
O'Reilly
Köln
2. Auflage, 2006
http://www.amazon.de/UML-2-0-kurz-gut-Pilone/dp/3897215217
ISBN: 3-89721-521-7

Harald Störrle
UML 2 für Studenten

Pearson Studium
Erscheinungsdatum: Auflage: 1 Mai 2005

ISBN: 3827371430

FOLDOC - Free-On-Line-Dictionary-Of-Computing

http://wombat.doc.ic.ac.uk/foldoc/

Eine Suche bei FOLDOC zu **formal methods** und **specification** ergibt folgendes:

Formale Methoden / formal methods
<Mathematik Spezifikation> Mathematisch basierte Technick zur Spezifikation, Entwicklung und Verifikation von Software und Harware Systemen.

Spezifikation / specification
<Jargon> Ein Dokument welches beschreibt, wie ein System arbeiten soll.

Benutzte UML2-/OCL-Tools

Hilfsmittel (Tools) zur formalen Spezifikation von OOP-Modellen mit Hilfe von OLC2:

Papyrus 1.11:
http://www.papyrusuml.org

oder in Kürze: http://wiki.eclipse.org/MDT/Papyrus

(Verfügbar (vorinstalliert) auf dem PI-Ausbildungscluster!)

Hinweis zur externen Benutzung des Ausbildungsclusters:
http://www.nomachine.com/download.php

Dazu installierte Serversoftware auf unserem Ausbildungscluster: FreeNX.

Einleitende Bemerkungen

Die „Object Constraint Language“
http://de.wikipedia.org/wiki/Object_Constraint_Language
und einige erste „Constraints“:

```
context Person inv: eltern->size() <= 2 ...
```

OMG-Dokumentation von OCL2: http://www.omg.org/docs/formal/06-05-01.pdf

- **Formale Methoden**

  sind logikbasierte Technicken für die
  Spezifikation,
  Entwicklung und
  Verifikation
  von SW- und Hardwaresystemen.

- Die **Spezifikation** eines Systems ist ein Dokument, das beschreibt, wie das System arbeiten soll.

- **Beispiele für entsprechende Beschreibungen:**

  a) Eine Funktion kann **implizit** (durch Angabe von Eigenschaften) spezifiziert werden:

  $max(s : \mathbb{N}_1\text{-set})m : \mathbb{N}_1$
  pre card $s \neq 0$
  post $m \in s \wedge \forall x \in s \cdot m \geq x$

  b) Eine Funktion kann **explizit** (durch Angabe eines Algorithmus) spezifiziert werden:

  $min(r : Real) : Real$
  post: if self <= r then result = self else result = r endif

- **Vor-/Nachbedingungen**

```
subSequence(lower : Integer, upper : Integer) : Sequence(T)

pre : 1 <= lower
pre : lower <= upper
pre : upper <= self->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
                                result->at(index - lower + 1)
                                   = self->at(index))
```

- **Beispiel zu Spezifikationsmängeln:**

  Euro-Panne bei der Deutschen Bank 24 (Update)

  Geldautomaten der Deutschen Bank 24 müssen sich wohl an den Euro erst noch gewöhnen. Wer Anfang Januar Euro-Beträge von Geldautomaten dieser Bank bezogen hat, durfte sich am heutigen Freitag wundern, dass ihm die Bank das 1,95-fache vom Konto abgebucht hat. Offensichtlich haben die Bank-Computer an Stelle der maßgeblichen Euro-Summe irrtümlich mit dem Zahlenwert des umgerechneten DM-Betrags gerechnet.

  Verunsicherte Kunden erfuhren zunächst nur, dass sogar die Angestellten der Bank dem Problem zum Opfer gefallen sind. Mit der Hoffnung auf hilfreichere Informationen mussten sie sich jedoch vorerst gedulden. Erst gegen elf Uhr konnten die Ansprechpartner an der Telefonhotline für etwas Beruhigung sorgen: "Das Problem ist bekannt, die falschen Buchungen werden automatisch zurückgezogen und korrigiert".

  Inzwischen fand die Bank heraus, dass bei einem nächtlichen Datenverarbeitungslauf einige Tausend der insgesamt etwa 1,5 Millionen angefallenenen Kontobewegungen durch einen Programmfehler falsch bearbeitet worden sind. Theoretisch hätten zwar auch herkömmliche Barabhebungen am Bankschalter betroffen sein können, doch zufällig drehte es sich bei den fehlerhaften Buchungen tatsächlich nur um Abhebungen von Geldautomaten, hieß es bei der Deutschen Bank 24. Das erklärt auch, warum bei anderen Banken, die gebührenfreies Abheben von denselben Geldautomaten wie die Deutsche Bank 24 ermöglichen, keine vergleichbaren Fehler aufgetreten sind.

  Markus Block, Sprecher der Deutschen Bank 24, erklärte gegenüber heise online, alle falschen Buchungen würden bis zum Samstag korrigiert sein, sodass kein Kunde finanzielle Nachteile zu erwarten habe. (hps/c't)

  Link zu diesem Artikel bei heise-online:
  http://www.heise.de/newsticker/meldung/23747

- **Ein Ausweg: Die Benutzung von, mit Einheiten versehenen Zahlenwerten, am Beispiel der Datei DM_Euro.cc**

Euro
− Wert : double
− Euro() + Euro(dw : DM) + Euro(e : const Euro &) + Euro(w : double) + ZeigeWert() : double

Abbildung 0.1: Die Klasse Euro

DM
− Wert : double
− DM() + DM(ew : Euro) + DM(d : const DM &) + DM(w : double) + ZeigeWert() : double

Abbildung 0.2: Die Klasse DM

- **Eine Anwendung:**

  Original mit anonymer Geldeinheit (double)

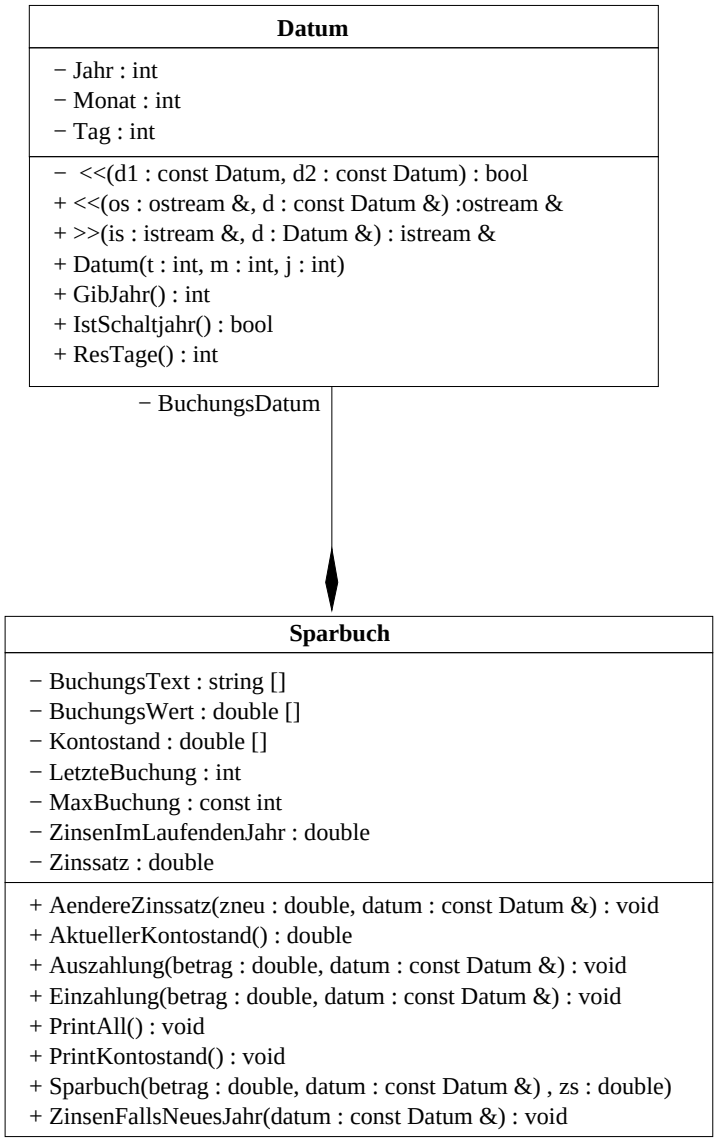


Abbildung 0.3: Die Klassen Datum und Sparbuch

Und besser:

Klasse Sparbuch mit Klasse DM und Klasse EURO:

```
//////////////////////////////////////
// Datei:    DM_Euro.cc
// Version:  1.1
// Zweck:    DM und Euro
// Autor:    Holger Arndt
// Datum:    23.05.2001
//////////////////////////////////////

#include <iostream>
#include <iomanip>

using namespace std;

class DM;

class Euro
{
private:
  double Wert;
public:
  Euro() : Wert(0.0) {};
  Euro(double w) : Wert(w) {};
  Euro(const Euro &e) : Wert(e.Wert) {};
  Euro(DM dw);
  double ZeigeWert() const { return Wert; };
};

class DM
{
private:
  double Wert;
public:
  DM() : Wert(0.0) {};
  DM(double w) : Wert(w) {};
  DM(const DM &d) : Wert(d.Wert) {};
  DM(Euro ew) : Wert(ew.ZeigeWert() * 1.95583) {};
  double ZeigeWert() const { return Wert; };
};

Euro::Euro(DM dw)
{
  Wert = dw.ZeigeWert() / 1.95583;
}
```

```
void DruckeEuroBetrag(const Euro &e)
{
  cout << "Geldbetrag: " << setiosflags(ios::fixed) <<
     setprecision(2)
       << e.ZeigeWert() << " Euro" << endl;
}

int main()
{
  Euro b1(12.3);
  Euro b2(14.12);
  DM b3(1.23);
  Euro b4;
  Euro b5(b1);

  DruckeEuroBetrag(b1);
  DruckeEuroBetrag(b2);
  DruckeEuroBetrag(b3);
  DruckeEuroBetrag(b4);
  DruckeEuroBetrag(b5);

  return 0;
}
```

- **Weitere Beispiele zu Spezifikationsmängeln:**

  - **PC-Problem lässt Walmart-Kunden in den USA dreifach zahlen**

    Ein Computer-Problem hat dazu geführt, dass 800.000 Karten-Transaktionen bei Walmart-Filialen in den ganzen USA doppelt oder dreifach verbucht wurden. Aufgetreten sei der Fehler beim Transaktions-Dienstleister First Data. US-Medien zitieren die First-Data-Sprecherin Staci Busby: "Die mehrfachen Mastercard- und Visa-Buchungen haben wir wieder zurückgenommen, vor Dienstag sind diese aber nicht ausgeführt. Jeder, der am 31. März bei Walmart eingekauft hat, sollte seine Abrechnung noch einmal überprüfen."

    Zu Details des Problems könne sie nichts sagen; klar sei jedoch, dass nur Walmart-Kunden davon berührt seien. Betroffene Kunden würden von First Data kontaktiert, versprach die Firmensprecherin, zudem sei eine kostenlose Info-Hotline geschaltet. (tol/c't)

    Link zu diesem Artikel bei heise-online:
    http://www.heise.de/newsticker/meldung/46278

  - **US-Bezahlsystem mit öffentlichen Kreditkartendaten**

    Durch einen primitiven Fehler auf den Webseiten des amerikanischen Bezahl-Dienstleisters PaySystems waren tausende von Kundendatensätzen einschließlich Kreditkartendaten zugänglich. Jeder PaySystems-Kunde konnte dabei die Daten anderer Kunden einsehen und sogar ändern.

    PaySystems bietet an, Bezahlvorgänge zu widerrufen. Dabei wird diesem Vorgang eine Transaktionsnummer zugewiesen, die beim Aufruf der zugehörigen Informationen als Parameter in der URL auftauchte. Durch Ändern dieses Parameters konnte man beliebige Transaktionen anderer Kunden abrufen und anschließend über eine zweite URL auch deren Adresse und Kreditkartendaten.

Besonders erschreckend war auch die Art und Weise, wie die Firma auf die Sicherheitslücke reagiert hat. Ein c't-Leser entdeckte das Problem zufällig und unterrichtete PaySystems unverzüglich. Als nach einer Woche nichts passierte, wendete er sich an heise Security. Auf unsere Nachfragen antwortete PaySystems prompt, dass man den Hinweis zur Kenntnis genommen habe und an der Beseitigung des Problems noch arbeite. Auf weitere Nachfragen, warum man die Seiten nicht unverzüglich gesperrt habe, kam keine Antwort mehr. Mittlerweile ist diese Lücke zwar geschlossen, aber die Daten standen – selbst nachdem PaySystems über das Problem informiert war – noch mindestens eine Woche ungeschützt im Netz.

Das Ausmaß des Problems lässt sich nur schwer abschätzen. Aber die Tatsache, dass die Transaktionsnummern sequenziell vergeben wurden und mehrere Stichproben sofort zum Erfolg führten, lässt darauf schließen, dass hunderttausende solcher Transaktionen zugänglich waren. Über welchen Zeitraum die Daten so offen im Netz standen, können wir nicht beurteilen. Nachdem PaySystems unsere diesbezüglichen Nachfragen ignoriert hat, rechnen wir nicht damit, dass der Dienstleister seine Kunden auf die mögliche Gefährdung der Kreditkartendaten hinweist. (ju/c't)

Link zu diesem Artikel bei heise-online:
http://www.heise.de/newsticker/meldung/45566

- **Report: Wurm Lovsan nicht Schuld an Blackout 2003**

  Eine amerikanisch-kanadische Untersuchungskommission der Energieaufsichtsbehörde (FERC) ist zu dem Ergebnis gekommen, dass der Wurm Lovsan/MSBlaster nicht der Verursacher des gigantischen Stromausfalls im Nordosten der USA im vergangenen Jahr war. Beim Blackout 2003 waren 50 Millionen Amerikaner zeitweise ohne Strom. Da zeitgleich der Wurm im Internet die Runde machte und Millionen von Windows-Rechnern infizierte oder lahmlegte, lag der Schluss nahe, Lovsan könne zum Ausfall beigetragen haben. Immerhin greifen Energieerzeuger schon seit längerem auf Windows für ihre Managementsysteme zurück. Anzeige

  Im Februar dieses Jahres wurde aber bekannt, dass ein Softwarefehler eines Unix-Systems zur Überwachung und Steuerung von Stromnetzen beim Erzeuger FirstEnergy den Ausfall begünstigte. Durch den Fehler wurden Alarme und Meldungen nicht mehr an das Kontrollpersonal weitergeleitet. Damit war es nicht mehr möglich, Gegenmaßnahmen zu ergreifen: Der Ausfall einer Versorgungsleitung führte zum Zusammenbruch des gesamten Stromverbundes.

  Der Fehler des Managementsystems sei aber laut Untersuchungsbericht weder auf Cyberattacken durch Al-Quaida noch durch Würmer oder Viren zurückzuführen. Grundlage der Ermittlungen waren Befragungen von Mitarbeitern, Telefonmitschnitte und Berichte von Behörden und Geheimdiensten. Allerdings habe man nicht die Logdateien von Netzwerkgeräten, Firewalls und Intrusion-Detection-Systemen ausgewertet, die eventuell tiefergehende Hin-

weise gegeben hätten.

Link zu diesem Artikel bei heise-online:
http://www.heise.de/newsticker/meldung/46328

- **Software-Fehler verursachte US-Stromausfall 2003**

  Acht Staaten im Nordosten der USA und Teile Kanadas blieben im August des vergangenen Jahres für fünf Tage ohne Strom. Insgesamt waren 50 Millionen Menschen betroffen. Schuld am Blackout war nach Angaben von SecurityFocus ein Softwarefehler des Managementsystems zur Überwachung und Steuerung von Stromnetzen beim Erzeuger FirstEnergy.

  Das betroffene System XA/21 stammt von General Electric und ist bei Erzeugern weit verbreitet. Der Fehler wurde nach einem mehrwöchigen intensiven Code-Audit gefunden und soll bisher nur beim großen Blackout aufgetreten sein. Nach Angaben des Sprechers von FirstEnergy löste eine besondere Kombination von Ereignissen und Alarmen den Fehler aus, woraufhin das System seinen Dienst einstellte. Der kurz darauf einspringende Backup-Server versagte ebenfalls, da er mit der Zahl der bereits aufgelaufenen, aber nicht verarbeiteten Meldungen überfordert war.

  In der Folge nahm das System auflaufende Alarme nicht mehr entgegen und meldete sie nicht an das Bedienpersonal weiter. Hinzu kam, dass den Betreibern nicht einmal auffiel, dass ihr System bereits versagt hatte. Eine Stunde lang soll die Kontrollstation veraltete Daten angezeigt haben. Bei auftretenden Störungen blieb zwangsläufig die Reaktion aus.

  Normalerweise koppelt ein Stromerzeuger sein Netz bei größeren Ausfällen von den anderen Stromnetzen ab, um weitere Schäden durch Überlast zu vermeiden. Somit bleibt ein Problem regional begrenzt. Da die Alarme aber nicht registiert wurden, reagierten die Operatoren nicht.

  FirstEnergy will nun seine XA/21-Systeme gegen die Produkte eines Wettbewerbers austauschen. Das North American Electric Reliability Council (NERC) hat eine Richtlinie herausgegeben, in der Maßnahmen beschrieben sind, Vorfälle wie am 14. August zu vermeiden. Unter anderem wird darin FirstEnergy aufgefordert, bis zum Austausch ihrer System alle notwendigen Patches für XA/21 zu installieren.

  Da sich der Zeitpunkt des Blackouts und der Ausbruch des Wurms Lovsan/Blaster überschnitten, gab es Vermutungen, der Wurm könnte den Ausfall verursacht haben. Auch warnte das CERT/CC Anfang August davor, dass Lovsan Unix-Systeme mit Distributed Computing Environment (DCE) angreift und zum Absturz bringen kann. XA/21 ist ein EMS/SCADA-System (Supervisory Control and Data Acquisition), das auf Unix mit X-Windows basiert. Sicherheitslücken gibt es hier reichlich. Somit kann zukünftig nicht ausgeschlossen werden, dass Würmer, die den Weg in ein Kontrollzentrum gefunden haben, solche Systeme beeinflussen können.

Link zu diesem Artikel bei heise-online:
http://www.heise.de/newsticker/meldung/44621

- **US-Sicherheitsexperten fordern bessere Ausbildung für Softwareentwickler**

  Die National Cyber Security Partnership (NCSP) fordert in ihrem aktuellem "Security Across the Software Development Life Cycle"eine bessere Ausbildung der Entwickler. Der Bericht befasst sich insbesondere mit dem Lebenzyklus von Software. Sicherheit müsse sich über die gesamte Lebensspanne eines Software-Produktes erstrecken. Jeder Abschnitt der Spanne, angefangen vom Design und Spezifikation, über die Implementierung und Tests bis hin zum Patch-Management soll unter den Gesichtspunkten der IT-Sicherheit bearbeitet werden.

  Die Arbeitsgruppe hat zur Definition entsprechender Empfehlungen vier Untergruppen gebildet, die sich mit Schulung von Entwicklern und Anwendern, Softwareprozessen und Patchen beschäftigen. Die vierte Gruppe – Incentive Subgroup – will ein Programm erarbeiten, um Herstellern das Entwickeln von sicherer Software schmackhaft zu machen. Dazu sollen Preisverleihungen und Zertifizierungen gehören. Daneben stellt man auch die Idee vor, die Sicherheit einzelner Softwaremodule als Messlatte für die weitere Karriere der jeweiligen Entwickler heranzuziehen.

  Die vergangenes Jahr gegründete Arbeitsgruppe hat sich die Verbesserung der Cyber Security der US-amerikanischen Informationsinfrastruktur zum Ziel gesetzt. Mitglieder sind diverse Sicherheitsexperten aus Forschung, Lehre und Industrie, sogar Vertreter der National Security Agency finden sich in der Gruppe. Die Vorsitzenden der Gruppe sind Ron Moritz von Computer Associates und Scott Charney von Microsoft. Ähnliche Ziele wie die NCSP verfolgen die Cyber Security Industry Alliance (CSIA) und der Global Council of CSOs

  Link zu diesem Artikel bei heise-online:
  http://www.heise.de/newsticker/meldung/46241

- **Softwarefehler plagt Mercedes-Diesel**

  Software-Bugs plagen die User nicht etwa nur, wenn sie vor dem Computer am Schreibtisch sitzen oder mit Mobilrechnern unterwegs sind. Internet-Zugang, Navigationsrechner oder multimediale Konsolen lassen das Auto zum IT-Problemfeld werden – darüber hinaus aber kämpfen Automobil-Elektroniker mittlerweile mit immer komplexeren computergestützten Steuerungssystemen und deren Software und damit auch mit den Bugs dieser Software. Jüngstes Beispiel: Wegen eines Softwarefehlers ruft DaimlerChrysler rund 10.000 Transporter der Mercedes-Benz-Modelle Vito und Viano mit Dieselmotoren zurück. In Deutschland sollen rund 3.000 Fahrzeuge betroffen sein.

> Ursache des Rückrufs ist ein Bug in der Software, mit der die Dieselsteuergeräte ausgerüstet sind. Sie aktivieren in Situationen, in denen dies eigentlich nicht vorkommen sollte, die Kraftstoffabschaltung, wodurch der Motor ausgehe. Betroffen seien Fahrzeuge mit Dieselmotoren, die zwischen November 2003 und April 2004 hergestellt wurden. Die Kunden würden durch die Servicestellen von Mercedes-Benz direkt angeschrieben, erklärte der Konzern; die Fahrzeuge erhielten eine fehlerbereinigten Software.
>
> Link zu diesem Artikel bei heise-online:
> http://www.heise.de/newsticker/meldung/48403

Weitere Links:

- Mars Climate Orbiter
- Fortress Programming Language: Seite 17, Seite 33f.
- Lufthansa-Check-in-Problem
- Computerpanne legt britische Flughäfen lahm
- Excel 2007 verrechnet sich beim Multiplizieren
- Software-Fehler

1 Rekapitulation: UML-Klassendiagramme

1.1 Klassen und Objekte

http://de.wikipedia.org/wiki/Klassendiagramm

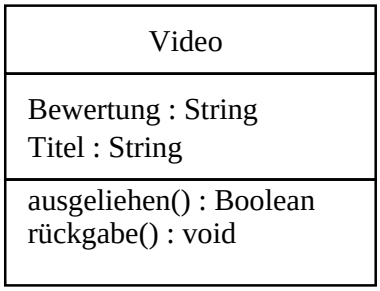


Abbildung 1.1: Eine Klasse

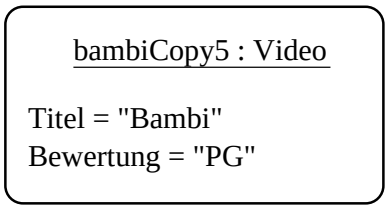


Abbildung 1.2: Ein Objekt dieser Klasse(Instanz)

```
<<primitive>> Datentypen:

Boolean
String
Integer
UnlimitedNatural
```

Siehe Kapitel 12 von http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/

1.1.1 Klassenbeschreibung

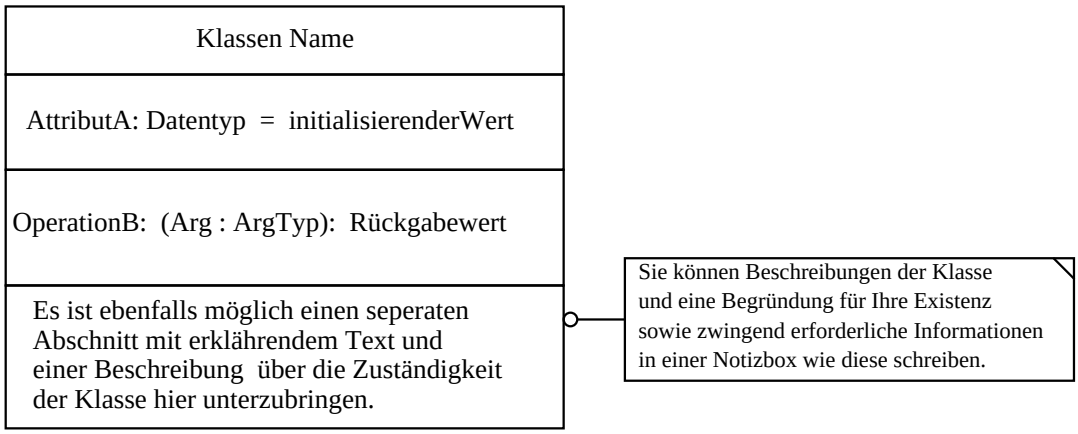


Abbildung 1.3: Beschreibung einer Klasse

Klassenname

Normale Schrift = konkrete Klasse
kursiveSchrift **oder** << abstract>> = abstrakte Klasse
(kursive Schriften sind nicht bildschirmfreundlich; benutzen Sie die Stereotyp-Notation)

Klassen- oder Instanzenattribute

Normale Schrift = Instanzen-Bereich
Unterstrichen **oder** $ = Klassenobjekte
($ ist kein UML-Standard)
Für abstrakte Methoden benutzen Sie = 0 oder << abstract>>
(0 ist kein UML-Standard)

Attribut- und Methodensichtbarkeit

+ public (öffentliche Sichtbarkeit)
- private (private Sichtbarkeit)
protected (geschützte Sichtbarkeit)

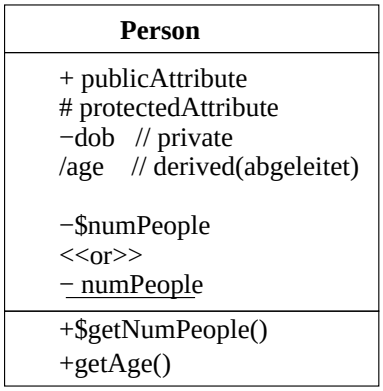


Abbildung 1.4: Eine Klasse: Person

- Das Attribut **age** ist hergeleitet.
- Die Anzahl der Instanzen der Klasse **Person** (**numPeople**) ist ein Attribut der Klasse **Person** selbst und nicht von einer Instanz der Klasse. Diese wird als statisches Klassen-Attribut (class static member variable) bezeichnet. Sie arbeitet wie eine globale Variable der Klasse. Manchmal wird als alternative Schreibweise für Klassenattribute und deren Verhalten das $ Zeichen verwendet.

1.1.2 Links und Assoziationen

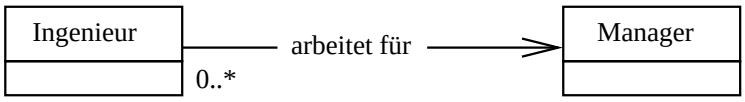


Abbildung 1.5: Assoziationen verbinden Klassenexemplare

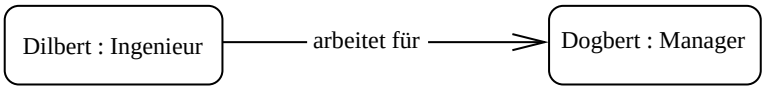


Abbildung 1.6: Assioziationen verbinden Klassenexamplare

1.1.3 Rollen und Assoziationsnamen

Rolle
: Benannte Instanzen einer Klasse die an das anderen Ende der Assoziation geschrieben werden, gewöhnlich ein Substantiv.

Assoziationsname
Benennt die Assoziation selbst; erfordern zuweilen einen Pfeil, der die Richtung der Assoziation anzeigt; gewöhnlich Verben oder Verbschlagworte.

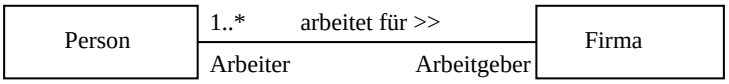


Abbildung 1.7: Rollen in Klassen

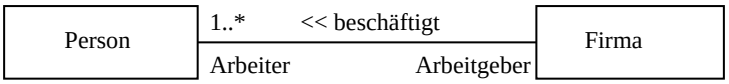


Abbildung 1.8: Rollen in Klassen (Fortsetzung)

1.1.4 Multiplizitäten (Kardinalitäten)

- Multiplizitäten beschreiben die Anzahl der Instanzen am Assoziationsende.
- Beispiele:

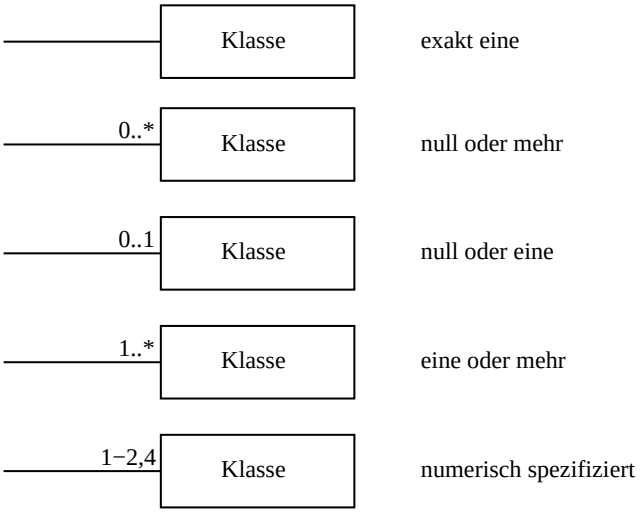


Abbildung 1.9: Multiplizität

Anmerkung: n und * können anstelle von 0..* verwendet werden.

1.1.5 Stereotypen

Stereotypen

Eine konventionelle Kategorisierung für modellierende Entitäten.

- Sie werden oft bei Klassen, Assoziationen und Methoden angewendet.
- Sie bieten einen Weg, UML zu erweitern; sie dienen zur Definition eigener, für spezielle Probleme modellierter Elemente.
- Einige Stereotypen werden von CASE-Werkzeugen (CASE tool generator) erkannt.

Es gibt zwei Wege, Stereotypen darzustellen:

- Benutzen Sie normale UML-Elemente, mit dem Stereotypnamen zwischen << und >>.
- Benutzen Sie eigene eindeutige Icons.

Beispiele:

```
<< abstract >>, << interface >>, << exception >>,
<< instantiates >>, << subsystem >>, << extends >>,
<< instance of >>, << friend >>,
<< constructor >>, << thread >>, << uses >>,
<< global >>, << create >>, << invent your own >>
```

1.1.6 Tagged Values

- Tagged Values sind ein weiterer Mechanismuss, UML zu erweitern: Er erlaubt es, dem Modell neue Eigenschaftsspezifikationen hinzuzufügen (Name = Wert).

Gebräuchliche Beispiele für **tagged values** sind:

- {Autor = (Dave,Ron)}
- {Versionsnummer = 3}
- {Location = d:\Location\uml\examples}
- {Location = Node: Middle Tier}

1.1.7 Generalisierung, Spezialisierung und Vererbung

- **Arbeitnehmer** generalisiert **Manager** und **Ingenieur**.
- **Ingenieur** spezialisiert **Arbeitnehmer**.
- **Manager** ist eine **Art/Sorte** von **Arbeitnehmer**.
- **Manager** und **Ingenieur** erben die Schnittstellen von **Arbeitnehmer** und in diesem Fall auch einige Implementierungseinzelheiten.

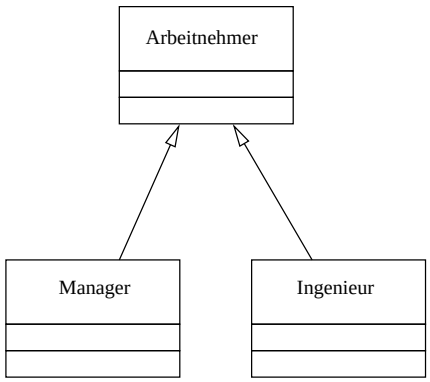


Abbildung 1.10: Generalisierung, Spezialisierung und Vererbung

1.2 Abstrakte Klassen

- Eine Generalisierung ohne vollständige Implementierungsspezifikation.
- Sie wird in UML mit dem Stereotyp `<< abstract >>` angezeigt.
- In C++ werden alle **pure virtual** Methoden = 0 deklariert.
- In Java wird sie mit dem Schlüsselwort "abstract"gekennzeichnet
- Ein **Interface** ist wie eine abstrakte Klasse, aber ohne jede Implementierung.

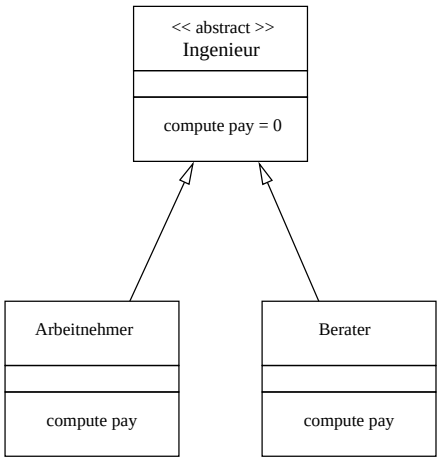


Abbildung 1.11: Abstrakte Klassen

1.3 Komposition / Aggregation

Das Rautenzeichen wir für verschiedene Eigenschaften / Konzepte eingesetzt.

- Teil- / Ganzes-Beziehung (am häufigsten verwendet)
- Hat - ein
- Hat - eine Sammlung - von
- Ist zusammengesetzt - aus

Beachten Sie, wie die Zeit die Kardinalitäten beeinflussen kann: Ein Auto kann viele Fahrer haben, aber zu einem bestimmten Zeitpunkt, kann es nur einer fahren.

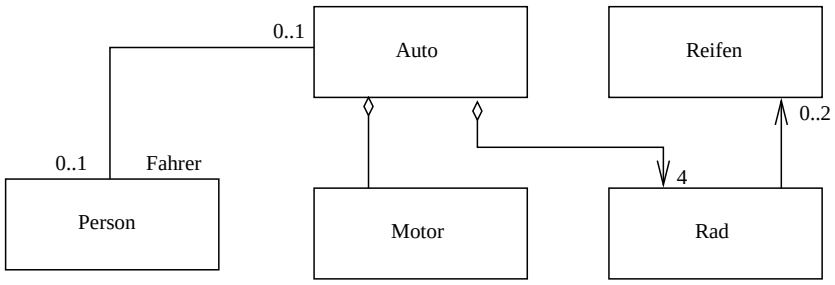


Abbildung 1.12: Komposition / Aggregation

Komposition:

- UML benutzt ein ausgefülltes Rautensymbol für eine **Komposition**.
- Das leere Rautensymbol beschreibt eine **Aggregation**.
- Eine **Komposition** ist eine stärkere Assoziation als eine **Aggregation**. Der Unterschied besteht darin, dass bei einer **Komposition**, ein Teil nie mehr als ein Ganzes ist und das ein Teil und ein Ganzes immer einen gemeinsamen Lebenszyklus/Lebenszeit haben.
- In folgenden Beispiel sind **Zeilen** ein fester und permanenter Bestandteil des **Layouts**, aber die Anzahl der Zeichen in jeder Zeile verändert sich zur Lebenszeit des Layout-Exemplars.

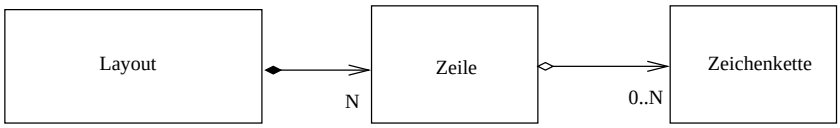
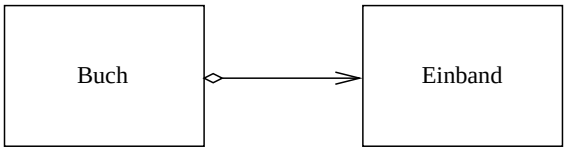


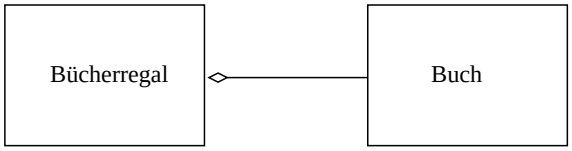
Abbildung 1.13: Komposition zwischen Layout und Zeile

- Das Objekt **Zeile** ist ein Teil vom Objekt **Layout**, sodass Zeilen erzeugt werden, wenn ein Layout erzeugt wird und Zeilen zerstört werden, wenn ein Layout zerstört wird. **Zeile** hat keine selbstständige Existenz.
- Beispiel: Ein Buch besteht aus Seiten (pages) und einem Einband (cover).



Aggregation:

- Instanzen der Klasse Buch existieren unabhängig von Objekt Bücherregal, aber Objekt Bücherregal hat Kentniss von seinen Instanzen der Klasse Buch.

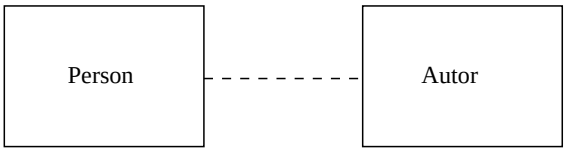


Assoziation:

- Ein Objekt der Klasse Buch hält eine halb-permanente Referenz zu einem Objekt der Klasse Autor ohne jede einschränkende Semantik.
- Beispiel: Bücher haben einen Autor

Dependancy:

- Instanzen der Klasse Person haben vorübergehende Beziehungen zu Instanzen der Klasse Autor
- Beispiel: Eine Person liest ein Buch, dann gibt sie es einem Freund.



1.4 Qualifizierte Assoziationen/Qualified Associations

- Sie werden benutzt, damit Instanzen einer Klasse, die in einer "ein zu viele"-Beziehung zu einer anderen Klasse B stehen, über einen eindeutigen Identifizierer schnell auf die Instanzen von B zugreifen zu können.
- Qualifizierte Assoziationen sind für gewöhnlich mit einer Art "Wörterbuch"ausgestattet (auch als assoziative Felder bekannt), etwa ein **Hash Table** oder einer **TreeMap**.
- Warum ist eine qualifizierte Assoziation ein besseres Modell?

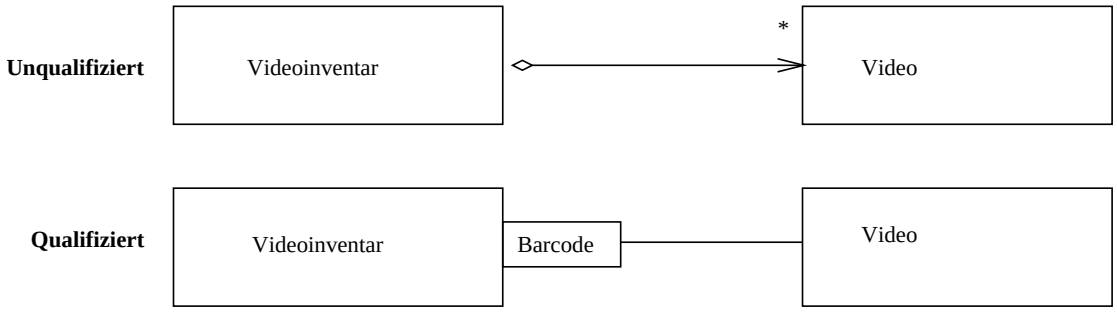


Abbildung 1.14: Qualifizierte Assoziation

1.5 Assoziazionsattribute

Attribute hängen manchal von zweie Objekten ab. Wenn ein solches Attribut viel komplexer ist als ein skalarer Wert, sollte es als eine eigene Klasse modelliert werden.

- Im folgenden Beispiel ist ein Arbeitsvertrag ein Attribut für die **"arbeitet für"**-Assoziation.
- **Anmerkung:** Die Semantik der Assoziationsklasse (so wie sie modelliert wurde) zeigt an, dass für jedes Personen/Firma-Paar, exakt ein Arbeitsvertrag existiert. Somit beschreibt dieses Modell, dass eine Person nicht zu zwei unterschiedlichen Zeiten für dieselbe Firma arbeiten kann.
- **Anmerkung:** Der Stereotyp `<<Geschichte>>` erklährt den Zeitaspekt der Beziehung: Er besagt, das eine Person über die Zeit für viele Firmen arbeiten kann, aber zu einer bestimmten Zeit immer nur für keine (0) oder eine (1) Firma arbeitet.

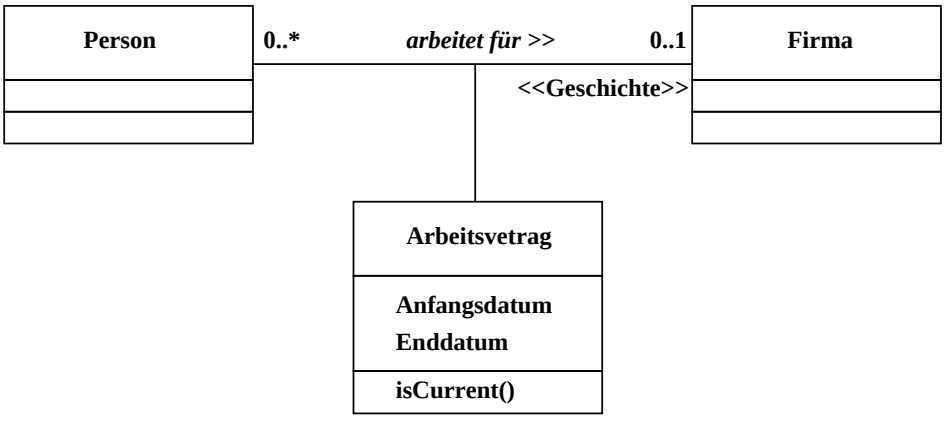


Abbildung 1.15: Assoziierte Attribute

- Die Semantik des Assoziatiationsattributes entspricht dem relationalen Datenbank-Design.
- In der Implementierung kann eine Person eine Reihe von Beschäftigungen aufnehmen. Jedes Beschäftigungsverhältniss kennt eine Person und eine Firma.
- Beachten Sie die Änderung in der assoziierten Kardinalität und die Tatsache das die "Arbeiter"-Beziehung nun abgeleitet ist (angezeigt wird die mit "/").

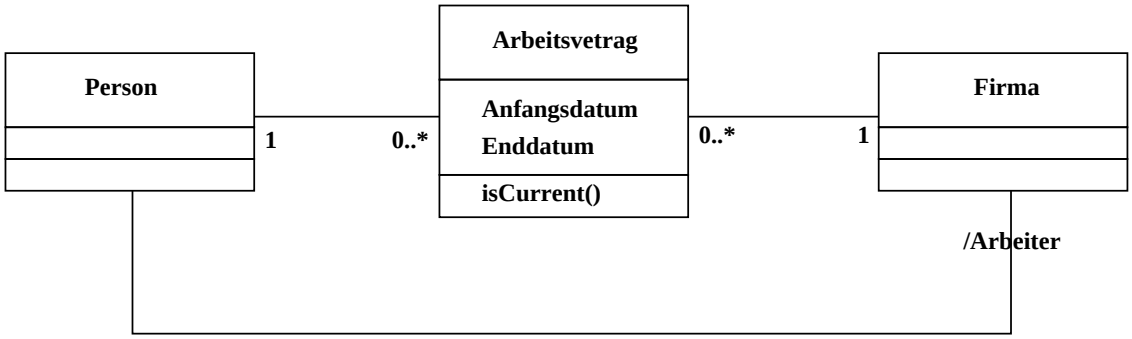


Abbildung 1.16: Assoziiertes Attribut (Fortseztung)

1.6 UML 2.0

UML2.1-Notationsübersicht

http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_classdiagram.html

http://www.jeckle.de/files/umltutorial.pdf (Seite 16-22)

Assoziationen im Klassendiagramm

1.7 Modell und Metamodell

UML User-Modell und Metamodell (Seite 9f., 19f.)

4-Schichten-Architektur von UML(Seite 10)

Forschungsministerium fördert Standard für IT-Sicherheit

2 OCL:

2.1 Spezifikation einfacher Klassen nach Prinzipien der SdV

Ein einfaches Beispiel:

VALUE , KEY

mydictionary

- keys: vector<KEY>*
- values: vector<VALUES>*
- count: unsigned int

```
/* <<basic queries>> */
+ get_count(): unsigned int
+ has(k: const KEY &) : bool
+ value_for (k: const KEY &): VALUE

/* << constructors >> */
+ << constructor >> mydictionary()
+ << constructor >> mydictionary(
        s: const mydictionary<KEY,VALUE>&)
+ ∽mydictionary():void

/* disable assignmet operator */
- = (s: const mydictionary(KEY,VALLUE &):mydictionary&

/* derived queries */
/* ... */

/* modificators */
+ put(h:const KEY &, ∨: const<VALUE>&):void
+remove(h:const KEY &):void
```

Klassifikation in

- grundlegende Abfragen (Queries/Observatoren)
- abgeleitete Abfragen (Queries/Observatoren)
- Aktionen (Modifikatoren)
- Konstruktoren/Destruktoren

Siehe dazu zum Beispiel:

Spezifikation durch Vertrag — eine Basistechnologie für eBusiness

2.2 Vor- und Nachbedingungen in OCL

OCL-Manual Seite 8f.

2.3 Programming by Contract

http://de.wikipedia.org/wiki/Design_by_contract
(**SdV**, *Design by Contract* [1], *Programming by Contract*) ist eine Methode zur Spezifikation der dynamischen Semantik von Softwarekomponenten mit Hilfe von Verträgen aus erweiterten boolschen Ausdrücken. **SdV** basiert auf der Theorie der abstrakten Datentypen und formalen Spezifikationsmethoden. Spezifizierte Komponenten können Module, Klassen oder Komponenten im Sinne von Komponententechnologien (wie Microsofts COM, .NET oder Suns EJB) sein. Verträge ergänzen das Kunden-Lieferanten-Modell:

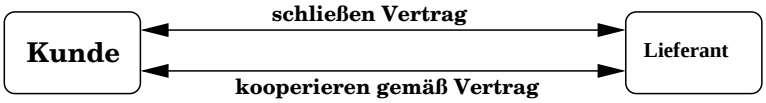


Abbildung 2.1: Kunden-Lieferanten-Modell

Grundlegend für die Vertragsmethode ist das **Prinzip der Trennung von Diensten in Abfragen und Aktionen** (*command-query separation)*:

- **Abfragen** geben Auskunft über den Zustand einer Komponente, verändern ihn aber nicht. Sie liefern als Ergebnis einen Wert. Die Abfragen einer Komponente beschreiben ihren abstrakten Zustand.

[1] „Design by Contrakt“ ist ein Warenzeichen von Interactive Software Engeneering.

- **Aktionen** verändern den Zustand einer Komponente, liefern aber kein Ergebnis. Die Aktionen einer Komponente bewirken ihre Zustandsveränderungen.

Diesem Prinzip folgend sind seiteneffektbehaftete Funktionen als Dienste zu vermeiden[2].

2.3.1 Methodenklassifikation in C++

- const-Methoden (Abfragen/Queries/Observatoren) teilt man in wesentliche und abgeleitete solche ein.
- Die wesentlichen Observatoren erlauben eine vollständige Spezifizierung des Zustands eines Klassenexemplars.
- Sie (und nur sie) werden nicht durch Nachbedingungen spezifiziert. Sie dienen vielmehr dazu, abgeleitete Observatoren und Modifikatoren (das sind nicht-const-Methoden) in ihren Nachbedingungen näher zu bestimmen.
- Dazu werden die abgeleiteten Observatoren durch eine Nachbedingung unter Benutzung einer oder mehrerer wesentlicher Observatoren spezifiziert.
- Modifikatoren werden durch eine Nachbedingung unter Benutzung aller wesentlicher Observatoren spezifiziert, um den exakten Zustand des Exemplars am Ende des Modifikatoraufrufs anzugeben.
- Verzichte (evtl.) in Nachbedingungen von Modifikatoren darauf, explizit zu spezifizieren, was sich nicht ändert (in der Annahme, dass alles nicht explizit genannte als *ungeändert* zu gelten hat). Leider ist nicht immer klar, was *ungeändert* zu bedeuten hat: Mindestens dann sollten Frameregeln (Rahmenbedingungen) explizit spezifizieren, was nach Aufruf des Modifikators *gleich* ist wie vorher.
- Explizite Spezifikation aller Rahmenbedingungen können bei programminterner Überprüfung der Nachbedingungen fehlerhafte Implementierungen aufdecken!
- Schreibe für jede Methode eine Vorbedingung mit Hilfe von
  - Abfragen und
  - Bedingungen an Methodenparameter.

  Hier (bei den Vorbedingungen) dürfen auch abgeleitete Abfragen, die eventuell effizienter sein können als eine sonst nötige Kombination mehrerer wesentlicher Abfragen, benutzt werden.
- Sorge dafür, dass bei Erfülltsein der Vorbedingungen auf jeden Fall die Nachbedingungen ebenfalls erfüllt sind (oder — in Ausnahmefällen — eine Exception ausgelöst wird).

[2] In bestimmten Fällen, z.B. bei Fabrikfunktionen, können Seiteneffekte sinnvoll sein. Solche Funktionen sind nicht als Spezifikatoren verwendbar und sollten entsprechend gekennzeichnet sein.

- Sorge dafür, dass die Abfragen in Vorbedingungen effizient berechnet werden (evtl. durch Hinzufügen weiterer effizienter abgeleiteter Abfragen). Vergesse nicht, die evtl. hinzugefügten neuen abgeleiteten Abfragen durch Nachbedingungen (und Vorbedingungen) zu spezifizieren.
- Nutze Invarianten um die Abhängigkeit von Abfragen zu spezifizieren (Konsistenzbeziehungen).
- Untersuche alle Abfragen paarweise auf Redundanzen und formuliere solche explizit als Invarianten.
- Wann immer Abfrage-Ergebnisse oder Methoden-Parameter eingeschränkte Wertebereiche besitzen, formuliere dies explizit in Form von
  - Vorbedingungen,
  - Nachbedingungen

  oder
  - Invarianten.
- Schreibe die Nachbedingungen von virtuellen (also überschreibbaren) Methoden immer in der Form
  `Vorbedingung implies Nachbedingung`
  (`Ensure((!Vorbedingung) || Nachbedingung)`), um die Redefinition in Kindklassen konfliktfrei zu ermöglichen.

2.3.2 Vertragspflichten/Vertragsnutzen

Ein Grund für die strikte Trennung von Abfragen und (reinen) Aktionen ist, dass Abfragen als **Spezifikatoren** dienen, d.h. als Elemente von Verträgen. **Verträge** setzen sich aus Bedingungen folgender Art zusammen:

- **Invarianten** einer Komponente sind allgemeine unveränderliche Konsistenzbedingungen an den Zustand einer Komponente, die vor und nach jedem Aufruf eines (public) Dienstes gelten. Formal sind Invarianten boolsche Ausdrücke über den Abfragen der Komponente; inhaltlich können sie z.B. Geschäftsregeln (busisness rules) ausdrücken.
- **Vorbedingungen** (preconditions) eines Dienstes sind Bedingungen, die vor dem Aufruf eines Dienstes erfüllt sein müssen, damit er ausführbar ist. Vorbedingungen sind boolsche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes.
- **Nachbedingungen** (postconditions) eines Dienstes sind Bedingungen, die nach dem Aufruf eines Dienstes erfüllt sind; sie beschreiben, welches Ergebnis ein Dienstaufruf liefert oder welchen Effekt er erzielt. Nachbedingungen sind boolsche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes,

erweitert um ein Gedächniskonstruckt, das die Werte von Ausdrücken vor dem Dienstaufruf liefert.

Verträge legen Pflichten und Nutzen für Kunden und Lieferanten fest. Die Verantwortlichkeiten sind klar verteilt:
Der Lieferant garantiert die Nachbedingung jedes Dienstes, den der Kunde aufruft, falls der Kunde die Vorbedingung erfüllt. Eine verletzte Vorbedingung ist ein Fehler des Kunden, eine verletzte Nachbedingung oder Invariante (bei erfüllter Vorbedingung) ist ein Fehler des Lieferanten. Die Verträge spezifizieren also eindeutig die Verantwortlichkeit bei Auftreten eines Fehlers.

	Kunde	Lieferant
Pflicht	Die Vorbedingung einhalten.	Die Nachbedingung herstellen und die Invariante erfüllen.
Nutzen	Ergebnisse/Wirkungen nicht prüfen, da sie durch die Nachbedingungen garantiert sind. Bei Methodenaktivierung werden die Anweisungen ausgeführt, die die Nachbedingungen herstellen und die Invarianten erhalten	Die Vorbedingungen nicht prüfen; sie sind durch den Vertrag garantiert und Mehrfachüberprüfungen sollten vermieden werden.

Tabelle 2.1: Pflichten - Nutzen von Kunden und Lieferanten

Schwache Vorbedingungen erleichtern den Kunden die Arbeit, starke Vorbedingungen dem Lieferanten. Je schwächer die Nachbedingungen sind, umso freier ist der Lieferant und umso ungewisser sind die Kunden über das Ergebnis/den Effekt. Je stärker die Nachbedingungen sind, umso mehr muß der Lieferant leisten.

2.3.3 Beispiele

Einige Beispielverträge für eine Klasse `vektor` (notiert in `nana`):

- friend-Funktion `Norm()` (abgeleitete Abfrage/Query/Observator)

```
double Norm(const vektor& v)
{
  REQUIRE(v.invariant());
  // ...
  // double qsum = ...
  ENSURE(approximatelyEqualTo(qsum, S(int k=v.lo(), k<=v.hi(),k++,
                                      v(k)*v(k)), 2.0));
  ENSURE(approximatelyEqualTo(result*result,qsum,2.0));
```

```
  return result;
}
```

- Methode `normalize()` (Modifikator ohne Rückgabewert (void))

```
void vektor::normalize()
DO
        REQUIRE(Norm(*this)!=0.0);
        ID(vektor value_old(*this));
        ...
        ENSURE(approximatelyEqualVekTo(result*n, value_old, 2.0));
        ENSURE(approximatelyEqualTo(Norm(result), 1.0, 2.0));
END
```

- i-ter Einheitsvktor (statische Klassenmethode)

```
vektor vektor::ei(int n, int i)
{
  REQUIRE((n>=1) && (1<=i) && (i<=n));
  ...
  ENSURE(result.lo()==1);
  ENSURE(result.hi()==n);
  ENSURE(E1(int k=result.lo(), k<=result.hi(), k++,
            result(k)!=0.0));
  ENSURE(result(i)==1.0);
  ENSURE(result.invariant());
  ...
}
```

- Konstruktor

```
vektor::vektor(const double x[], int n) : low(1), high(n)
{
  REQUIRE((n>=1) && (x!=0));
  REQUIRE("x[] hat mindestens n Komponenten");
  ...
  ENSURE(lo()==1 && hi()==n);
  ENSURE(A(int k=lo(), k<=hi(), k++, (*this)(k)==x[k-lo()]));
END
```

- Modifikator

```
void vektor::changeValueAt(int i, double x)
DO
  REQUIRE((lo()<=i) && (i <=hi()));
  ...
  ENSURE((*this)(i)==x);
  ENSURE("alle anderen Komponenten von *this ungeaendert");
END
```

  Überlegen Sie sich einen expliziten Nichtänderungsvertrag für „alle anderen Komponenten“ von `*this` (Frame-Bedingung).

- `operator!=` (abgeleitete Abfrage)

```
bool vektor::operator!=(const vektor& w) const
DO
  REQUIRE(w.invariant());
  ...
  ENSURE(result == ((hi()-lo())!=(w.hi()-w.lo())) ||
     E(int k=lo(), k<=hi(), k++, (*this)(k)!=w(k-lo()+w.lo())));
  ...
}
```

2.3.4 Subcontracting

Es gelten folgende Regeln bei der Vererbung (von is-a-Methoden):

a) Vorbedingungen können in einer Kindklasse abgeschwächt werden oder müssen gleich sein.

b) Nachbedingungen in einer Kindklasse müssen gleich oder stärker sein als diejenigen der Elterklasse.

c) Invarianten in der Kindklasse müssen ebenfalls gleich oder stärker als in der Elterklasse sein.

Dann ist ein echtes *Subcontracting* realisiert.

Bemerkung: Es reicht die Kindnachbedingung im Falle des Eintreffens der Eltervorbedingung gleich oder stärker als die Elternachbedingung zu realisieren. Im Falle „Kindvorbedingung **and not** Eltervorbedingung“ darf die Kindnachbedingung frei gewählt werden.

$$
\begin{array}{l}
Invariante_{Kindklasse} = Invarinte_{Elterklasse} \wedge ... \\
Vorbedingung_{Kindmethode} = Vorbedingung_{Eltermethode} \vee ... \\
Nachbedingung_{Kindmethode} = \begin{cases} Nachbedingung_{Eltermethode} \wedge ... & \text{, falls } Vorbedingung_{Eltermethode} \\ \text{beliebig} & \text{, sonst} \end{cases}
\end{array}
$$

Ein Beispiel mit Contract/Subcontract in nana:

```
class name_list{
...
public:

  /////////// basic queries:

  unsigned int get_count() const;   // number of items in stack

  bool has(const string& a_name) const;
...
/////////// (pure) modificators:

  virtual void put(const string& a_name); // Push a_name into list
}

void name_list::put(const string& a_name)    // Push a_name into list
DO
   REQUIRE(/* name not in list */    !has(a_name));
   ID(set<string> contents_old(begin(),end()));
   ID(int count_old = get_count());
   ID(bool not_in_list = !has(a_name));
...
   ENSURE(has(a_name));
   ENSURE( (!not_in_list) || (get_count() == count_old + 1));
   ID(set<string> contents(begin(),end()));
   ENSURE( (!not_in_list) || (contents == contents_old + a_name));
END

...
///////////////// child class relaxed_name_list ///////////
////////////////   (more user friendly)   ////////////////

class relaxed_name_list : public name_list{
    ///////// (pure) modificators: (redefined)
```

```
    virtual void put(const string& a_name);   // Push a_name into list
...
}
void relaxed_name_list::put(const string& a_name)    // Push a_name into list
DO
   REQUIRE(/* nothing */  true);       // usable without conditions
   ID(set<string> contents_old(begin(),end()));
   ID(int count_old = get_count());
   ID(bool not_in_list = !has(a_name));
...
   ENSURE(has(a_name));
   ENSURE((!not_in_list) || (get_count() == count_old + 1));  // &&
   ENSURE(  not_in_list  || (get_count() == count_old));
   ID(set<string> contents(begin(),end()));
   ENSURE(  not_in_list  || (contents == contents_old));
   ENSURE((!not_in_list) || (contents == contents_old + a_name));
END
...
///////////////// child class relaxed_name_list ///////////
////////////////   (more user friendly)   ////////////////

class relaxed_name_list : public name_list{
    ///////// (pure) modificators: (redefined)

    virtual void put(const string& a_name);   // Push a_name into list
...
}
void relaxed_name_list::put(const string& a_name)    // Push a_name into list
DO
   REQUIRE(/* nothing */  true);       // usable without conditions
   ID(set<string> contents_old(begin(),end()));
   ID(int count_old = get_count());
   ID(bool not_in_list = !has(a_name));
...
   ENSURE(has(a_name));
   ENSURE((!not_in_list) || (get_count() == count_old + 1));  // &&
   ENSURE(  not_in_list  || (get_count() == count_old));
   ID(set<string> contents(begin(),end()));
   ENSURE(  not_in_list  || (contents == contents_old));
   ENSURE((!not_in_list) || (contents == contents_old + a_name));
END
```

2.3.5 Contracting — Zusammenfaßung

PbC	Verpflichtungen	Vorteile
Benutzer der Klasse	delegiert nur bei erfüllter Vorbedingung	kommt in den Genuß der garantierten Nachbedingung und Invarianten
Anbieter der Klasse	(nur bei gültiger Vorbedingung:) muß die Nachbedingung erfüllen	braucht Vorbedingung nicht überprüfen; kann sich auf deren Einhaltung verlassen

Tabelle 2.2: Verpflichtungen/Vorteile von Verträgen zwischen Komponentenanbieter und -benutzer

Klassen-Invarianten (Gültige Attributwertkombinationen)

- schränken Werte von Attributen ein, trennen gültige von ungültigen Exemplaren einer Klasse

- spezifizieren Redundanzen (vgl. Day/Month/Year, Count/IsEmpty, ...)

Methoden-Vorbedingungen (an Attribute und Parameter)

- schränken den Bereich ein, in dem die Methode erfolgreich sein muß, benutzt werden darf

Methoden-Nachbedingungen (an Attribute und Parameter)

- spezifizieren (formal) das Ergebnis der Methode (das **was**, nicht das **wie**)

Was vor und nach jeder Methode gelten muß (in Form von Hoare-Tripeln notiert):

Konstruktor:
$\{VB_{Parameter}\}$ Konstruktor $\{Inv \wedge NB_{Konstruktor}\}$

Destruktor:
{Inv} Destruktor {-}

Jede andere (öffentliche) Methode M:
$\{VB_M \wedge Inv\}$ M $\{NB_M \wedge Inv\}$

aus: http://www.cse.yorku.ca/course_archive/2004-05/F/3311/sectionA/22-InheritDBCgen.pdf

2.3.6 Weitere Subcontracting-Beispiele

2.3.6.1 Funktion invert

Beispiel in Eifel:

Ursprüngliche Definition:
invert(epsilon:REAL) is - - Invert martix with precision epsilon
    require epsilon>= 10^(−6)
    ...
    ensure abs ((Current * inverse) - Identy) <= epsilon
end

Redefinition:
invert(epsilon:REAL)is - - Invert matrix with precision epsilon
    require else epsilon>= 10^(−20)
...
    ensure abs ((Current * inverse) - Identy) <= (epsilon/2)
end

2.3.6.2 Interface Directory

Ein Vertrag zwischen Kunde und Unternehmer (in Cleo spezifiziert) laute:

```
INTERFACE Directory[Keys, Values]
ACTIONS
   Put(IN k:Keys, IN v:Values)
      PRE
         NOT Has(k)
      POST
         Has(k)
         ValueFor(k)= v
         Count = OLD(Count)+1
.
.
.
```

Kann er durch den Unternehmer allein nicht zeitgerecht erfüllt werden, so kann sich dieser eventuell folgendermaßen aus seiner Notlage befreien: Ein anderer Unternehmer biete den folgenden Vertrag an:

```
INTERFACE Directory[Keys, Values]
ACTIONS
   Put(IN k:Keys, IN v:Values)
```

```
      PRE
         TRUE
      POST
         Has(k)
         ValueFor(k)= v
         NOT OLD(Has(k)) IMPLIES Count = OLD(Count)+1
         OLD(Has(k))     IMPLIES Cout = OLD(Count)
.
.
.
```

2.3.6.3 Interface LoeseLGS

```
------------------------------- LoeseLGS-Elter
ACTIONS
   LoeseLGS( IN A : Matrix,
             IN b : Vektor,
            OUT x : Vektor )
      PRE
         NOT Det(A) = 0
      POST
         || A * x - b || <= EPSILON
```

sowie ein Subcontract:

```
------------------------------- LoeseLGS-Kind
ACTIONS
   LoeseLGS( IN A : Matrix,
             IN b : Vektor,
            OUT x : Vektor )
      PRE
         TRUE
      POST
         NOT Det(A) = 0 IMPLIES || A * x - b || <= EPSILON
         Det(A) = 0     IMPLIES "x ist eine Minimalstelle von || A * x - b ||"
```

2.3.6.4 Interface Bruecke

```
INTERFACE Fussgaengerbruecke
------------------------------ Fussgaengerbruecke
   QUERIES
      MaxLast : REAL
      AktLast : REAL
   INVARIANTS
      MaxLast >= 7500
      AktLast <= MaxLast
   ACTIONS
      ueberquereBruecke( IN gew : REAL,
                         OUT Guthaben : INTEGER )
         PRE
            gew  + AktLast <= MaxLast
            gew <= 200
            Guthaben >= 2
         POST
            AktLast = OLD(AktLast) + gew
            Guthaben = OLD(Guthaben) - 2
      verlasseBruecke( IN gew : REAL )
         ...
```

sowie ein Subcontract:

```
INTERFACE Autobruecke
------------------------------ Autobruecke
   QUERIES
      MaxLast : REAL
      AktLast : REAL
   INVARIANTS
      MaxLast >= 800000
      AktLast <= MaxLast
   ACTIONS
      ueberquereBruecke( IN gew : REAL,
                         OUT Guthaben : INTEGER )
         PRE
            gew  + AktLast <= MaxLast
            gew <= 20000
            Guthaben >= 20
         POST
            AktLast = OLD(AktLast) + gew
            OLD(gew) <= 200     IMPLIES Guthaben = OLD(Guthaben) - 2
            NOT OLD(gew) <= 200 IMPLIES Guthaben = OLD(Guthaben) - 20
      verlasseBruecke( IN gew : REAL )
```

. . .

Aufgabe:

Überlege Contracts und Subcontracts im Umfeld:
— Kunde/Stammkunde
— Firmenkonto/Privatkundenkonto
— Vereinsmitglied /Vorstandsmitglied
— ...

2.3.7 Zusammenfaßung der SdV-Prinzipien

1. Observatoren (und nur diese) haben einen Ergebniswert; sie ändern den Obje ktinhalt nicht!
   Modifikatoren haben **keinen** Ergebniswert.

2. Unterscheide: "grundlegende Observatoren" von

3. "abgeleiteten Observatoren". Jeder abgeleitete Observator hat eine Nachbedingung, die auf die grundlegenden Observatoren zurückgreift.

4. Für jeden Konstruktor/Modifikator schreibe eine Nachbedingung, die die Wer te aller brundlegenden Observatoren am Ende einer Methode festlegt.

5. Für jeden Observator und jeden Konstruktor/Modifikator schreibe notwendige Vorbedingungen.

6. Schreibe für jede Klasse eine Invariante, die die sich nicht ändernden Mer kmale der Objekte beschreibt (also **gültige** und **ungültige** Obje kte unterscheidet).

7. Die grundlegenden Observatoren sind ein minimaler Methodensatz, der dazu dient den Zustand eines Exemplars vollständig zu charakterisieren. Sie haben außer Konsistenzbeziehungen zu anderen Methoden **keine** Nachbedingungen.

2.4 Ein OCL2-Vertrag

```
package mydictionary

context mydictionary
inv: count >= 0
inv: keys->size() = values->size()
inv: keys->size() = count

/* basic observators */

context mydictionary::get_count() : Integer
body: count

context mydictionary::has(k : KEY) : Boolean
post consistentWithCount: get_count()=0 implies not result
/* post: get_count() = KEY->select(k | has(k))->size() */

context mydictionary::value_for(k: KEY): VALUE
pre: has(k)

/* constructor */

context mydictionary::mydictionary()
post: get_count() = 0


context mydictionary::mydictionary(s : mydictionary) // const &
post: s.get_count() = self.get_count()
/* post: KEY->forAll(k | s.has(k) implies s.value_for(k) = self.value_for(k)) */

/* modificator */

context mydictionary::put(k: KEY, v: VALUE)
pre: not has(k)
post: has(k)
post: get_count() = get_count@pre() + 1
post: value_for(k) = v
/* post: KEY->forAll(kl | has@pre(kl) implies value_for@pre(kl) = value_for(kl)) */

context mydictionary::remove(k: KEY)
pre: has(k)
post: not has(k)
post: get_count() = get_count@pre() - 1
```

```
/* post: KEY->forAll(kl | has@pre(kl) implies (kl = k or
                                                value_for@pre(kl) = value_for(kl))) *

endpackage -- mydictionary
```

2.5 Prinzipien der SdV

http://en.wikipedia.org/wiki/Design_by_contract

Jedem Unterprogramm werden Vorbedingungen (preconditions) und Nachbedingungen (postconditions) zugeordnet. Die Vorbedingungen legen fest, unter welchen Umständen das Unterprogramm aufrufbar sein soll. Beispielsweise darf ein Unterprogramm zum Lesen aus einer Datei nur dann aufgerufen werden, wenn die Datei vorher erfolgreich geöffnet wurde. Die Nachbedingungen legen die Bedingungen fest, die nach Abschluss des Unterprogrammaufrufs gegeben sein müssen.

Vor- und Nachbedingungen werden als boolesche Ausdrücke formuliert. Ist eine Vorbedingung nicht erfüllt (d. h. ihre Auswertung ergibt false, also „nicht zutreffend“), liegt ein Fehler im aufrufenden Code vor: Dort hätte dafür gesorgt werden müssen, dass die Vorbedingung erfüllt ist. Ist eine Nachbedingung nicht erfüllt, liegt ein Fehler im Unterprogramm selbst vor: Das Unterprogramm hätte dafür sorgen müssen, dass die Nachbedingung erfüllt ist.

Vor- und Nachbedingung bilden daher eine Art Vertrag (englisch contract): wenn der aufrufende Code die Vorbedingung erfüllt, dann ist das Unterprogramm verpflichtet, die Nachbedingung zu erfüllen.

Eine Invariante ist eine Aussage, die über die Ausführung bestimmter Programmbefehle hinweg gilt. Sie ist also vor und nach diesen Befehlen wahr, sie ist demnach nicht veränderlich, also invariant. Invarianten können zum Beweis der Korrektheit von Algorithmen verwendet werden und spielen eine große Rolle im Design By Contract. Dabei werden für eine Methode einer Schnittstelle deren Vor- und Nachbedingungen und alle Invarianten in ihrem Ablauf beschrieben. Mittels so genannter Assertions (Zusicherungen) kann man dieses Konzept implementieren, sofern es die verwendete Programmiersprache oder API unterstützt.

Eine Klasseninvariante schheidet gültige von üngültigen Objekten einer Klasse.

2.6 Ein Beispiel: Die Klasse java.awt.Color

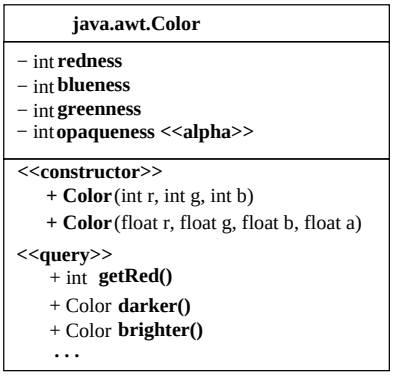


Abbildung 2.2: Die Standard Farbklasse: java.awt.Color

Was sagt Ihnen dieses Klassendiagramm? Was sagt es nicht?

2.6.1 Klassenspezifikation: java.awt.Color

Invarianten: (Für jedes Farbobjekt, c)
0<=redness(c)<=255 **and** 0<=greenness(c)<=255 **and**
0<=blueness(c)<=255 **and** 0<=opaqueness(c)<=255

Construktor Methoden:

Listing 2.1: Konstruktor-Methoden:

```
public Color(int r, int g, int b)
  pre: 0 <= r <= 255 und 0 <= g <= 255 und 0 <= b <= 255
                  --(throws IllegalArgumentException)
  -- modifies: redness, greenness, blueness, opaqueness
  post: redness == r und greenness == g und blueness == b
     und opaqueness == 255

public Color(float r, float g, float b, float a)
  pre: 0.0 <= r <=1 .0 und 0.0 <= g <= 1.0 und 0.0 <= b <=
     1.0 und 0.0 <= a <= 1.0
                  --(throws IllegalArgumentException)
  post: redness == r*255 und greenness == g*255 und
     blueness == b*255 und
  opaqueness == a*255
```

Query Methoden und Modifikatoren:

Listing 2.2: Query-Methoden

```
public int getRed()
 post: result == redness

public Color darker()
 post: result.redness == redness*0.7
   and result.greenness == greenness*0.7
   and result.blueness == blueness*0.7
   and result.opaqueness == 255

public Color brighter()
 post: (redness/ 0.7)>255 implies result.redness == 255
   and (redness/ 0.7)<=255 implies result.redness ==
      redness/ 0.7
   and (greenness/ 0.7)>255 implies result.greenness == 255
   and (greenness/ 0.7)<=255 implies result.greenness ==
      greenness/ 0.7
   and (blueness/ 0.7)>255 implies result.blueness == 255
   and (blueness/ 0.7)<=255 implies result.blueness ==
      bluness/ 0.7
   and result.opaqueness == 255
...
```

Bemerkung: Im Sinne des „Programming by Contract“ sollte der wesentliche Teil der Spezifikation „völlig“ implementierungsunabhängig durchgeführt werden.
Das heißt:

- kein Zugriff auf private Attribute bzw. Methoden
- keine Vorwegnahme der zu benutzenden Algorithmen
- Alle Vor- und Nachbedingungen sollten mit Hilfe der basic Queries arbeiten.

Bemerkung: Die Spezifikation mittels OCL geschieht aber nicht **nur** für den benutzenden Programmierer, sondern auch als Hilfe für das Implementierungsteam. **Hier** sollte natürlich auch auf implementierungsabhängige Einzelheiten Bezug genommen werden können. Z.Bsp. sollten die **basic Queries** selbst mittels Nachbedingungen spezifiziert werden.

2.7 Hinweise

1. Spezifiziere wo immer nötig implementirungsspezifische Entscheidungen (meist in der Form <> 0)

2. Stelle sicher, daß die in den Vorbedingungen benutzten Observatoren „effizient“ arbeiten. Falls nötig, füge zusätzliche abgeleitete schnell arbeitende Observatoren hinzu (virtuelle, nur zur Spezifikation benötigte Methoden), die in Ihren Nachbedingungen die aufwendigen Observatoren ersetzen können.

3. Wenn ein abgeleiteter Observator als Attribut implementiert wird, sollte die Klasseninvariante entsprechend erweitert werden.

4. Um die Neuimplementierung virtueller Methoden zu unterstützen sollte **jede** Nachbedingung einer virtuellen Methode durch Ihre Vorbedingung abgeschirmt sein.

Zu C++:

- Konstante Methoden sind (reine) Observatoren
- nichtkonstante Methoden sollten keine Ergebnisse liefern; bei Beendigung sollte die Klasseninvariante erfüllt sein
- Direkter modifizierender Zugriff auf Attribute ist gefährlich (warum?) und sollte deshalb nicht erlaubt sein.

2.8 OCL-Spezifikation von Klasseninterdependenzen

2.8.1 size() aller assoziierten Exemplare

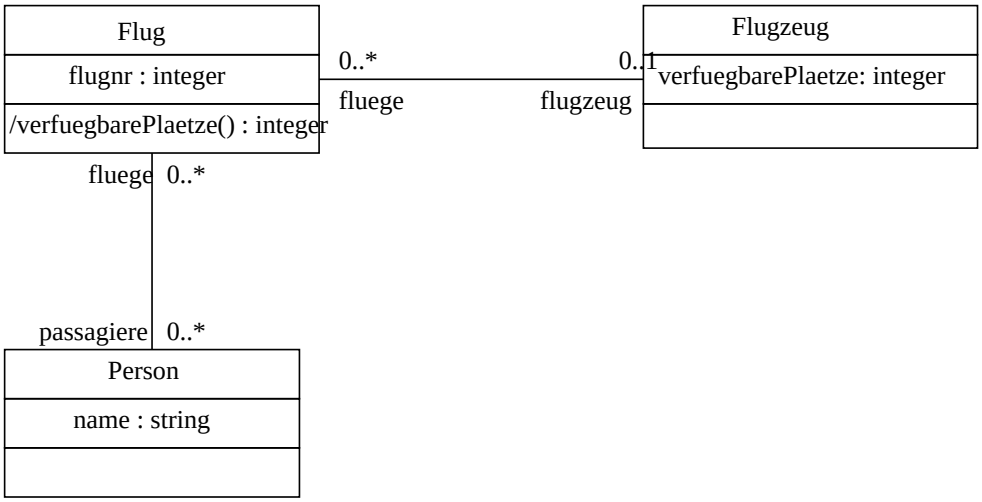


Abbildung 2.3: size() aller assoziierten Exemplare

z.B. Implementiert durch:

```
class Flug{
Flugzeug* flugzeug;
int      Flugnummer;
vector <Person *>
   passagiere;
int verfuegbarePlaetze();
}
```

```
class Person{
string name;
vector <Flug *> fluege;
}
```

```
class Flugzeug{
vector <Flug *> fluege;
int verfuegbarePlaetze;
}
```

Abbildung 2.4: Implementierungsbeispiel

Momentaner Status etwa:

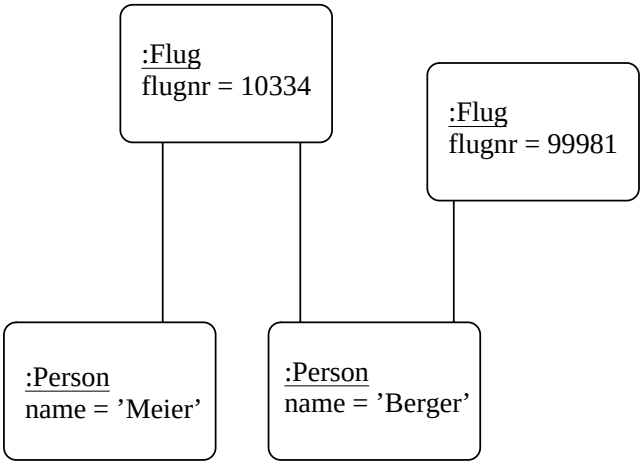


Abbildung 2.5: Zustand/Schnappschuß (Objektdiagramm)

Einschränkungen:

a) **context** Flug
inv: passagiere → size()<= flugzeug.verfuegbarePlaetze

b) context Flug :: verfuegbarePlaetze() : integer
body: flugzeug.verfuegbarePlaetze - passagiere → size()

2.8.2 includes() und forAll()

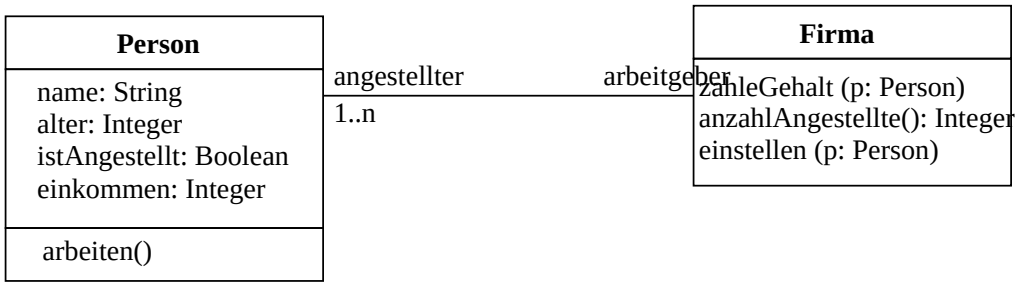


Abbildung 2.6: Modell Person-Firma

Zu diesem Klassendiagramm liege der folgende, in OCL formulierte Vertrag vor:

```
context  Person

  inv:   alter > 0

context  Person::arbeiten()
  pre:   istAngestellt

context  Firma::zahlteGehalt(p: Person)
  pre:   angestellter->includes(p)

context  Firma::anzahlAngestellte() : Integer
  pre:   angestellter->forAll(p | p.arbeitgeber = Set{self})
  post:  result = angestellter->size()

context  Firma::einstellen (p: Person)
  pre:   (p.alter > 13) and not p.istAngestellt
  post:  (angestellter->size()) = angestellter@pre->size() + 1)
          and p.istAngestellt

context  Person
  inv:   if istAngestellt then
               einkommen >= 300
         else
               einkommen < 300
         endif
```

or	if -
and	then -
xor	else -
not	endif
=	
<>	
implies	

Tabelle 2.3: logische Operationen in der OCL

2.8.3 Assoziationsklassen

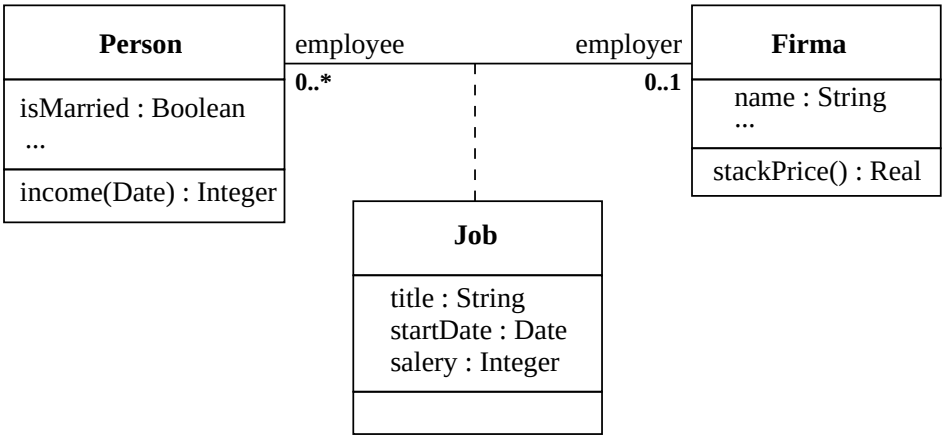


Abbildung 2.7: Includes

ist (automatisch) implementiert als:

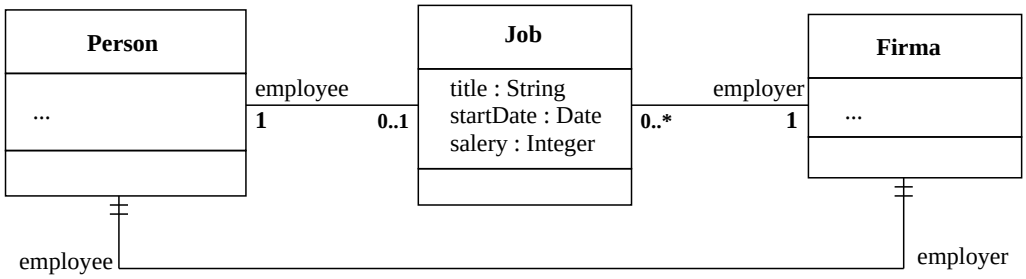


Abbildung 2.8: Includes

Hier ist zu benutzen:

context Firma
... job.employee ...

statt original

context Firma
... employee ...

Überlegen Sie sich eine abgeleitete Methode `employee` in Klasse `Firma` und analoges für die Klasse `Person`.

Weitere OCL-Ausdrücke:

context Person :: income (d:Date) : Integer
body self.job → Select(d >= startDate).salery → sum()

2.8.4 Qualifizierte Assoziationen

Abbildung 2.9: qualifizierte Aggregation

context Bank
inv: self.customer[100245].name ='Maier'

context Bank
inv: self.customer→exists(name='Otto')

2.8.5 Andere Methoden für die Collection Set

(im **context** Flugzeug)

Beispiele für Flugnummern:

$\underbrace{fluege}_{set(Flug)} \rightarrow \text{one}(flugnr = 123)$		Boolean
	true $\Longleftrightarrow$	Set(Flug) enthält genau einen Flug mit der FlugNr. 113
$\underbrace{fluege}_{set(Flug)} \rightarrow \text{isUnique}(flugnr)$		Boolean
	true $\Longleftrightarrow$	Jeder Flug in **fluege** hat eine andere FlugNr als jeder andere Flug in **fluege**!

fluege→ size()	UnlimitedNatural
fluege→ isEmpty()	Boolean
fluege→ notEmpty()	Boolean
fluege→ includes(Berlin_NewYork) ↑ von Typ Flug	Boolean
fluege→ includesAll(Berlin_NN) ↑ von Typ set(Flug)	Boolean
fluege→ count(Berlin_NewYork)	Integer
fluege→ excludes (Berlin_NewYork)	Boolean
fluege→ excludesAll (Berlin_NN)	Boolean
fluege.verfuegbarePlaetze()→ sum()	Integer, Real, ...
=	Boolean
<>	Boolean
-	Differenzmenge
a → intersection (b)	Durchschnitt
a → union(b)	Vereinigungsmenge
a → symmetricDifference(b)	Menge aller Elemente in A oder B, aber nicht in beiden, mathematisch: $(A \cup B) \setminus (A \cap B)$
flatten()	rekursives Entpacken von verschachtelten Mengen

Tabelle 2.4: Methoden für die Collection Set

2.8.6 Schleifen und Iteratoren

(im **context** Flugzeug)

fluege→ exists(flugnr=12)	Boolean
fluege.passagiere→ one(name='Meier')	Boolean
fluege→ forAll (verfuegbarePlaetze() <= 3)	Boolean
fluege→ select(verfuegbarePlaetze() > 0)	Collection
fluege→ reject (verfuegbarePlaetze() < 10)	Collection
fluege→ any(verfuegbarePlaetze() > 0)	ein Element
fluege.verfuegbarePlaetze()	{1,3,10, ...}
fluege→ collect (verfuegbarePlaetze()) (collectNested (expr))	{1,3,10, ...}
fluege→ isUnique (flugnr)	Boolean
fluege→ sortedBy (flugnr)	liefert: orderedSet oder Sequences
Set{1,2,3}→ iterate(i;sum:Integer=0 | sum+i)	
Set{1,2,3}→ iterate(i:Integer; sum:Integer = 0 | sum+i)	

Tabelle 2.5: Schleifen und Iteratoren

2.8.7 Andere Collections

	non ordered	ordered
Element kann nur einfach vorhanden sein	Set	OrderedSet
Element kann mehrfach vorhanden sein	Bag	Sequence

Operation	Set	OrderedSet	Bag	Sequence
=	X	X	X	X
<>	X	X	X	X
–	X	X	-	-
append(object)	-	X	-	X
as Bag()	X	X	X	X
asOrderedSet()	X	X	X	X
asSequence()	X	X	X	X
asSet()	X	X	X	X
at(index)	-	X	-	X
excluding(object)	X	X	X	X
first()	-	X	-	X
flatten()	X	X	X	X
including(object)	X	X	X	X
indexOf(object)	-	X	-	X
insertAt(index, object)	-	X	-	X
intersection(coll)	X	-	X	-
last()	-	X	-	X
prepend(object)	-	X	-	X
subOrderedSet(lower,upper)	-	X	-	-
subSequence(lower, upper)	-	-	-	X
symetricDifference(coll)	X	-	-	-
union(coll)	X	X	X	X

Tabelle 2.6: Collection Operationen mit verschiedenen Bedeutungen

Die genaue Bedeutung von zum Beispiel `excluding(object)`, `indexOf(object)`, `intersection(coll)`, `prepend()` und `symmetricDifference(coll)`lesen Sie bitte im Handbuch http://www.omg.org/spec/OCL/2.0/PDF (Nachbedingungen der Collection-Operationen) nach!

2.8.8 Together und automatische Code-Erzeugung

Listing 2.3: Klasse Bank

```
/*    Generated by Together    */

# ifndef BANK_H
# define BANK_H

class Bank {
private:

    /**
     * @associates Person
     * @supplierQualifier customer
     * @clientCardinality 0..1
     * @supplierCardinality 0..*
     * @undirected
     * @bidirectional Person#bank
     * @clientQualifier bank
     */
    integer accountNumber;
    map < integer, Person * > customer;
};
#endif //BANK_H
```

Listing 2.4: Klasse Bank

```
/*    Generated by Together    */

# ifndef GENDER_H
# define GENDER_H

/** @stereotype enumeration */
enum Gender {
    male, female
};
'endif //GENDER_H
```

Listing 2.5: Klasse Bank

```
/*    Generated by Together    */

# ifndef PERSON_H
# define PERSON_H
class Bank;
```

```
class Company;

class Person {
public:

    Integer income(Date);

private:
    Boolean isUnemployed;
    Date birthDate;
    Integer age;
    String firstName;
    String lastName;

    /**
     * @clientCardinality 1
     * @link association
     */
Gender sex;
Boolean isMarried;
    /**
     * @supplierCardinality 0..*
     * @supplierQualifier managedCompanies
     * @clientCardinality 1
     * @clientQualifier manager
     * @undirected
     * @bidirectional Company#manager
     */
//Company * linkCompany;
vector < Company * > managedCompanies;

    /**
     * @supplierCardinality 0..*
     * @supplierQualifier employer
     * @clientCardinality 0..*
     * @supplierQualifier employee
     * @associationAsClass Job
     * @undirected
     * @bidirectional Company#employee
     */
vector < Company * > employer;

    /**
     * @bidirectional Person#wife
```

```
     * @clientCardinality 0..1
     * @supplierQualifier husband
     * @supplierQualifier wife
     * @associationAsClass Marriage
     */
Person * husband;
/** bidirectional */
Person * wife;

Person * husband;
/** bidirectional */
Bank * bank;
};
#endif //PERSON_H
```

2.8.9 Fallstudie: Person/Haus/Hypothek

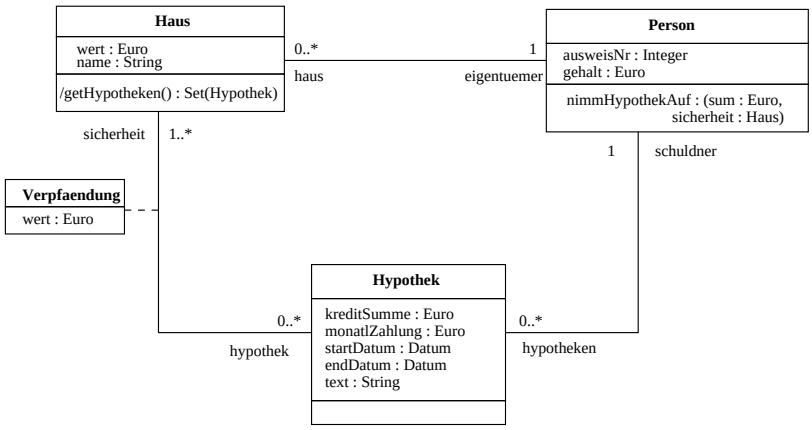


Abbildung 2.10: Klassendiagramm Hypothek

context Hypothek
inv: sicherheit.eigentuemer=Bag{schuldner}

context Hypothek
inv: startDatum < endDatum

context Person
def: anzahlHypotheken : Integer = $\underbrace{\underbrace{haus}_{Set(Haus)} .hypothek}_{Bag(Hypothek)} \rightarrow asSet() \rightarrow size()$

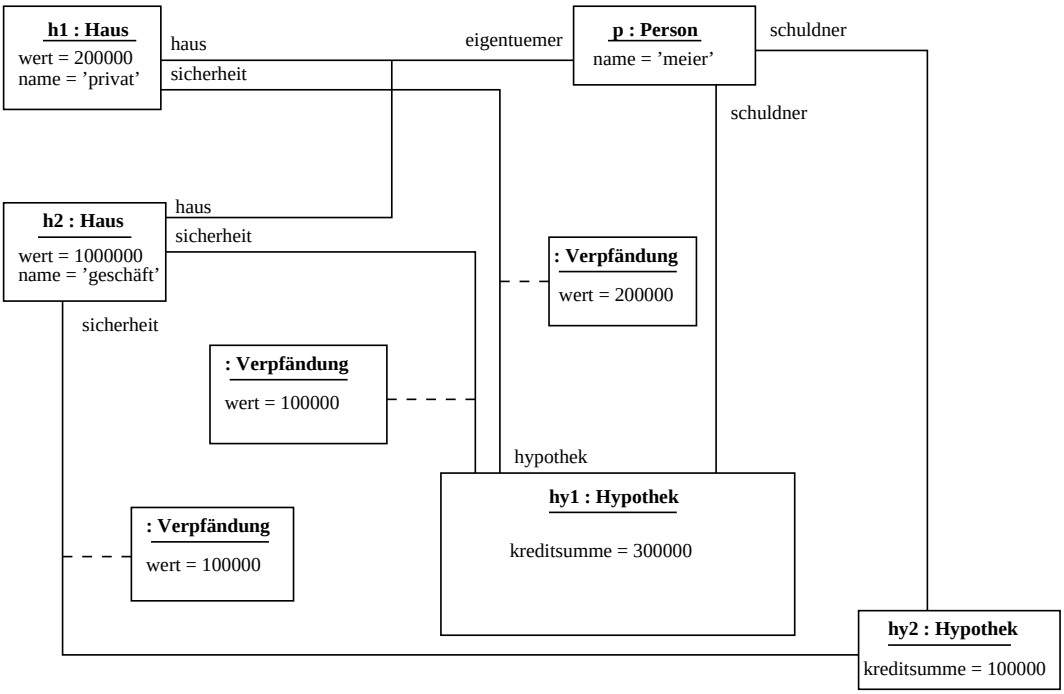
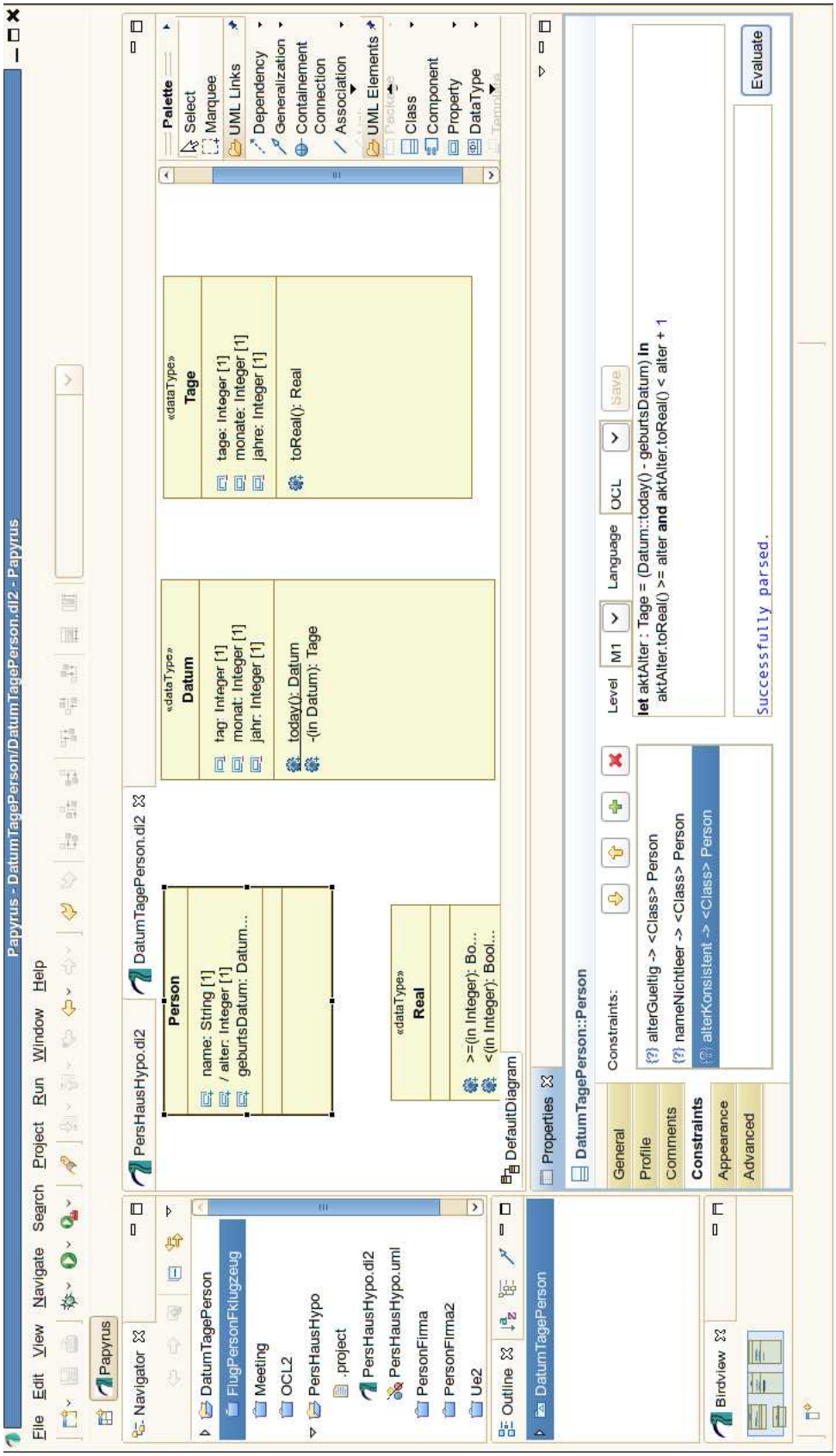


Abbildung 2.11: Hypothek mit zwei Häusern

p.haus= Set{h1,h2} mit **h1.hypothek**=Set{hy1}
h2.hypothek=Set{hy1,hy2}
p.haus.hypothek = Bag {hy1,hy1,hy2}
p.haus.hypothek → **as Set()**= {hy1,hy2}

2.8.10 Einige erste Hilfskomponenten



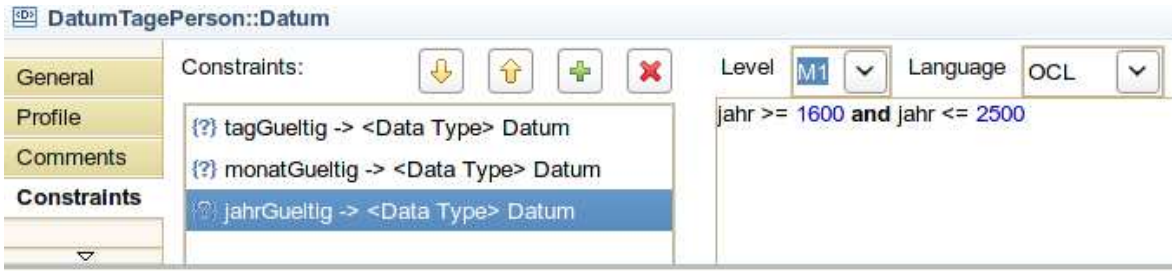
Listing 2.6: OCL-Constraints Datum

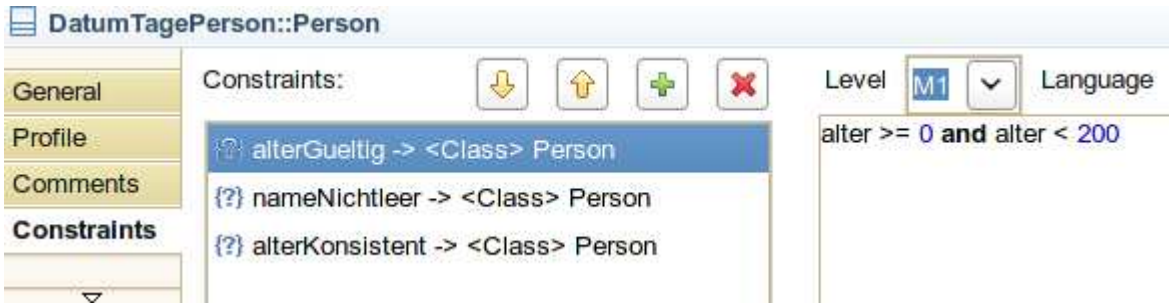
```
package core

context Datum

inv tagGueltig: tag >= 1 and tag <= 31
inv: monat >=1 and monat <= 12
inv: jahr >= 1600 and jahr <= 2500

endpackage -- core
```

Listing 2.7: OCL-Constraints Person

```
package core

context Person

inv: alter >= 0
inv: alter < 200
inv: name <> ''
inv: let aktAlter : Tage = (Datum::today() - geburtsDatum) in
    aktAlter.toReal() >= alter and aktAlter.toReal() < alter + 1

endpackage -- core
```

und ihre Benutzung in Papyrus:


DatumTagePerson::Datum
General
Profile
Comments
Constraints
Constraints:
Level
M1
Language
OCL
{?} tagGueltig -> <Data Type> Datum
{?} monatGueltig -> <Data Type> Datum
{?} jahrGueltig -> <Data Type> Datum
jahr >= 1600 and jahr <= 2500


DatumTagePerson::Person
General
Profile
Comments
Constraints
Constraints:
Level
M1
Language
{?} alterGueltig -> <Class> Person
{?} nameNichtleer -> <Class> Person
{?} alterKonsistent -> <Class> Person
alter >= 0 and alter < 200


DatumTagePerson::Person
General
Profile
Comments
Constraints
Constraints:
Level
M1
Language
OCL
Save
{?} alterGueltig -> <Class> Person
{?} nameNichtleer -> <Class> Person
{?} alterKonsistent -> <Class> Person
let aktAlter : Tage = (Datum::today() - geburtsDatum) in
aktAlter.toReal() >= alter and aktAlter.toReal() < alter + 1

2.8.11 OCL-Navigation durch UML-Modelle

objekt.rollenname

- ergibt das assoziierte Exemplar, falls die Multiplizität von `rollenname` 1 ist,
- den OrderedSet der assoziierten Exemplare, falls eine höhere Multiplizität vorliegt und die Rolle als `{ordered}` gekennzeichnet ist,
- den Set der assoziierten Exemplare sonst.

Falls UML keinen Rollennamen benutzt und das Assiziationsende eindeutig ist, so ist der kleingeschriebene Name der Zielklasse zu benutzen.

Ist das vorletzte Glied einer Navigationskette eine Collection und der letzte Rollenname als `{ordered}` gekennzeichnet, so liegt eine Sequence ansonsten ein Bag vor.

Eine explizite oder impliziete `collect()`-Operation einer Collection erzeugt einen Bag beziehungsweise eine Sequence.

2.8.12 Alle Instanzen einer Klasse: allInstances()

Die Methode

allInstances():**Set**(T)

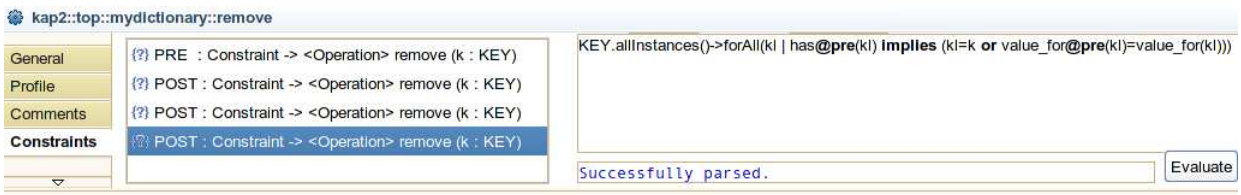
liefert alle (endlich viele) Instanzen einer User-definierten Klasse. Sie darf nicht benutzt werden für String, Integer und Real.

Ein Beispiel:

```
context Person
inv uniqueNames: person.allInstances()->forAll(p1,p2|
                                p1<>p2 implies p1.name <>p2.name)
```

Weitere Beispiele:
proglang.informatik.uni-freiburg.de/teaching/swt/2009/v11-ocl.en.pdf (Seite 24/26)

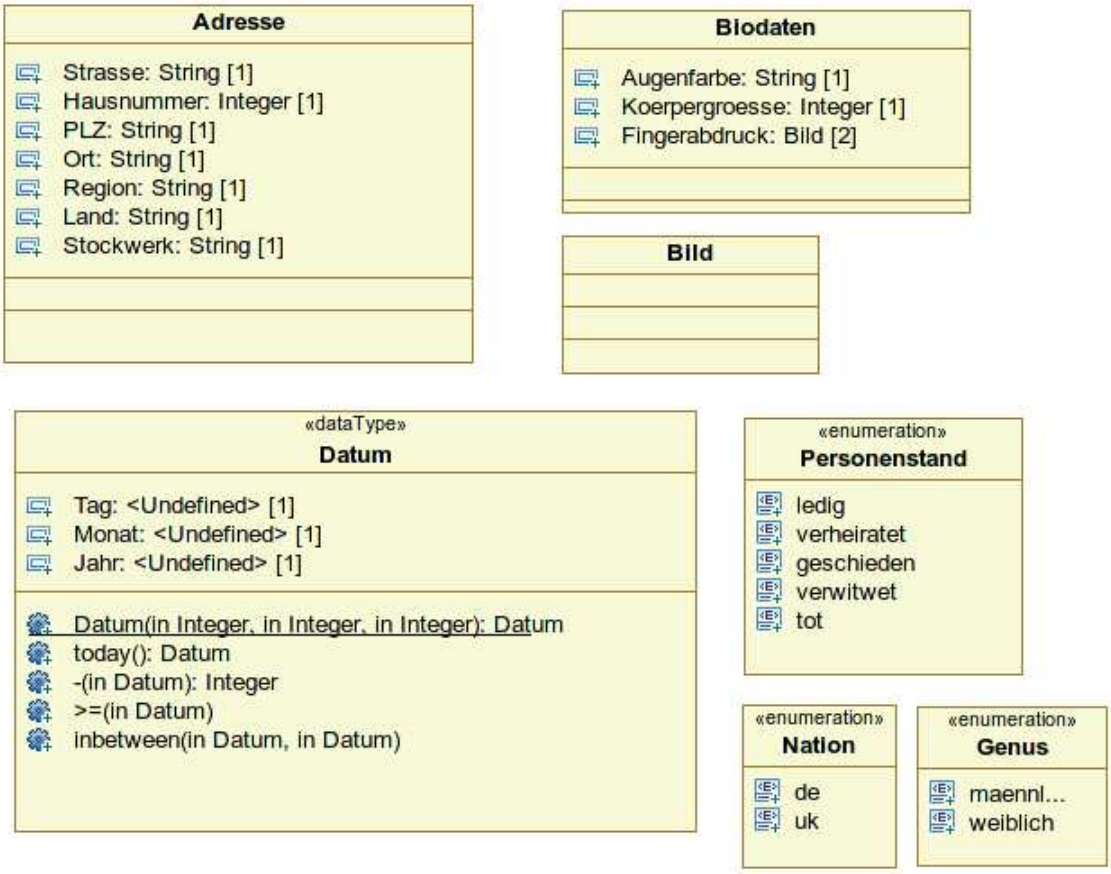
Damit ist unsere `mydictionary`-Spezifikation endgültig realisierbar:



2.8.13 Software-Pannen

http://www.heise.de/newsticker/meldung/Software-Panne-Chaos-am-Hauptbahnhof-Hannover-887996.html
http://www.heise.de/newsticker/meldung/Ministerin-sieht-EC-Karten-Probleme-noch-nicht-geloest-900326.html
http://www.heise.de/security/meldung/EC-Karten-Probleme-bei-Haendlern-in-Deutschland-geloest-899827.html

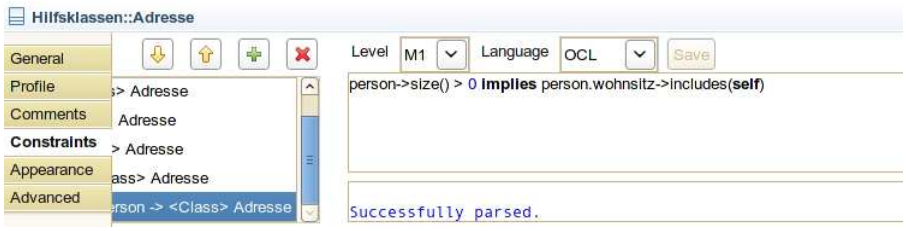
2.8.14 Hilfsklassen: Adresse, BioDaten, Datum, Personenstand, Nation, Genus



Listing 2.8: Constraints Adresse

```
package core

context Adresse
inv : Hausnummer > 0
inv : Land <> ''
inv : Ort <> ''
inv : PLZ.toInteger() > 0 and PLZ.toInteger() <100000
inv : Strasse <> ''
```

```
inv : Person->size() > 0 implies Person.Wohnsitz->includes(self
    )

context Adresse::Land
init : 'Deutschland'

context Adresse::Region
init : 'NRW'

context Adresse::Person
init : Set{}

endpackage
```



Listing 2.9: Constraints Biodaten

```
package core

context BioDaten
inv: Augenfarbe <> ''

context BioDaten::Augenfarbe
init : 'unbekannt'

endpackage -- core
```

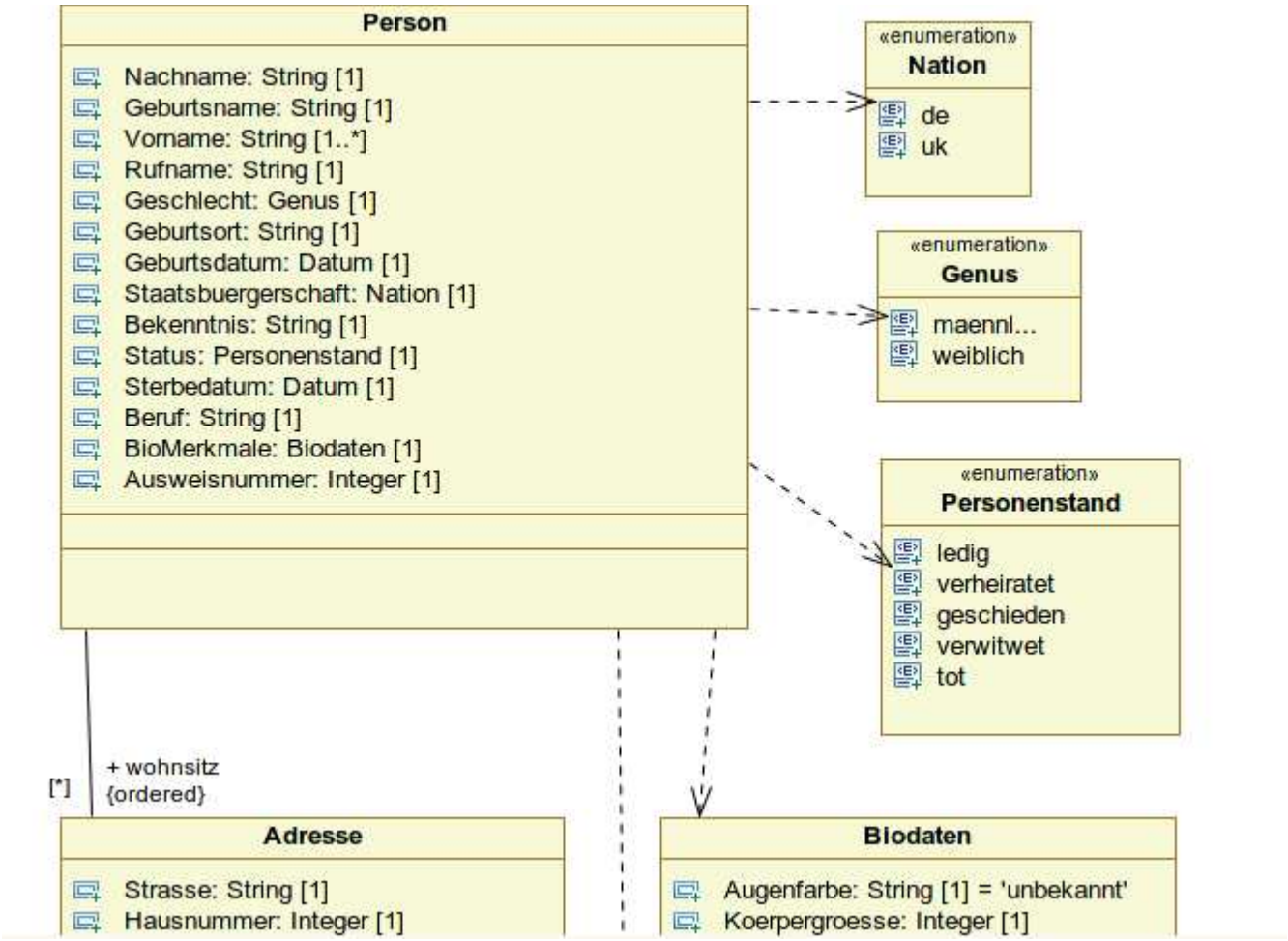
Listing 2.10: Constraints Datum

```
package core

context Datum          -- vituelle Methode = Hilfsmethode
def : gueltigesDatum( t : Integer, m : Integer, j : Integer ) :
    Boolean =
      1920 <= j and j <= 2100 and
      1 <= m and m <= 12 and
      1 <= t and
      t <=
```

```
            if      Set{4, 6, 9, 11}->includes(m)
              then 30
            else if  Set{1, 3, 5, 7, 8, 10, 12}->includes(m)
              then 31
            else if ((j.mod(4)=0) and
                       not(j.mod(100)=0)) or
                    (j.mod(400)= 0)
                                then 29 -- Schaltjahr
                                else 28
            endif
            endif
            endif

context Datum
inv : gueltigesDatum(Tag, Monat, Jahr)

context Datum::Datum( t : Integer, m : Integer, j : Integer ) :
    Datum
pre  : gueltigesDatum(t, m, j)
post : result.oclIsNew()
post : result.Tag = t
post : result.Monat = m
post : result.Jahr = j

/* Problem context Datum::-() wird nicht akzeptiert , deshalb
   Workaround: */
context Datum::minus( d : Datum ) : Integer
pre : self >= d
post : if Monat > d.Monat then result = Jahr - d.Jahr
       else if Monat < d.Monat then result = Jahr -d.Jahr - 1
       else -- Monat = d.Monat
            if Tag >= d.Tag then result = Jahr - d.Jahr
            else result = Jahr - d.Jahr - 1
            endif
       endif
       endif

/* analoges Workaround; */
context Datum::groesserGleich(d:Datum) : Boolean
post : if Jahr > d.Jahr then result = true
      else if Jahr < d.Jahr then result = false
      else -- Jahr = d.Jahr
            if Monat > d.Monat then result = true
            else if Monat < d.Monat then result = false
```

```
            else -- Monat = d.Monat
                if Tag >= d.Tag then result = true
                else result = false
                endif
            endif
            endif
        endif
        endif

context Datum::inbetween( from : Datum, to : Datum ) : Boolean
pre : to >= from
body : (to >= self) and (self >= from)

context Datum   -- virtuelles Attribut = Hilfsattribut
def : invalidDatum : Datum =
    Datum::Datum(31, 12, 2100) -- 1. Versuch eines opt. DTs

endpackage
```

2.8.15 Person zur Modellierung von Personenstandsdaten



Listing 2.11: Constraints Person

```
package core

context Person
inv : elter->size() <=2
inv : Ausweisnummer >= 0    /* 0 bei fehlendem Ausweis */
inv : Wohnsitz->size() > 0 implies Wohnsitz.Person->includes(
   self)
inv : Muendel->notEmpty() implies Muendel.Vormund->asSet() =
   Set{self}
inv : Vormund->notEmpty() implies Vormund.Muendel->includes(
   self)
inv : elter->notEmpty() implies elter.kind->includes(self)
inv : kind->notEmpty() implies kind.elter->includes(self)
inv : ehemann->notEmpty() implies ehemann.ehefrau->includes(
```

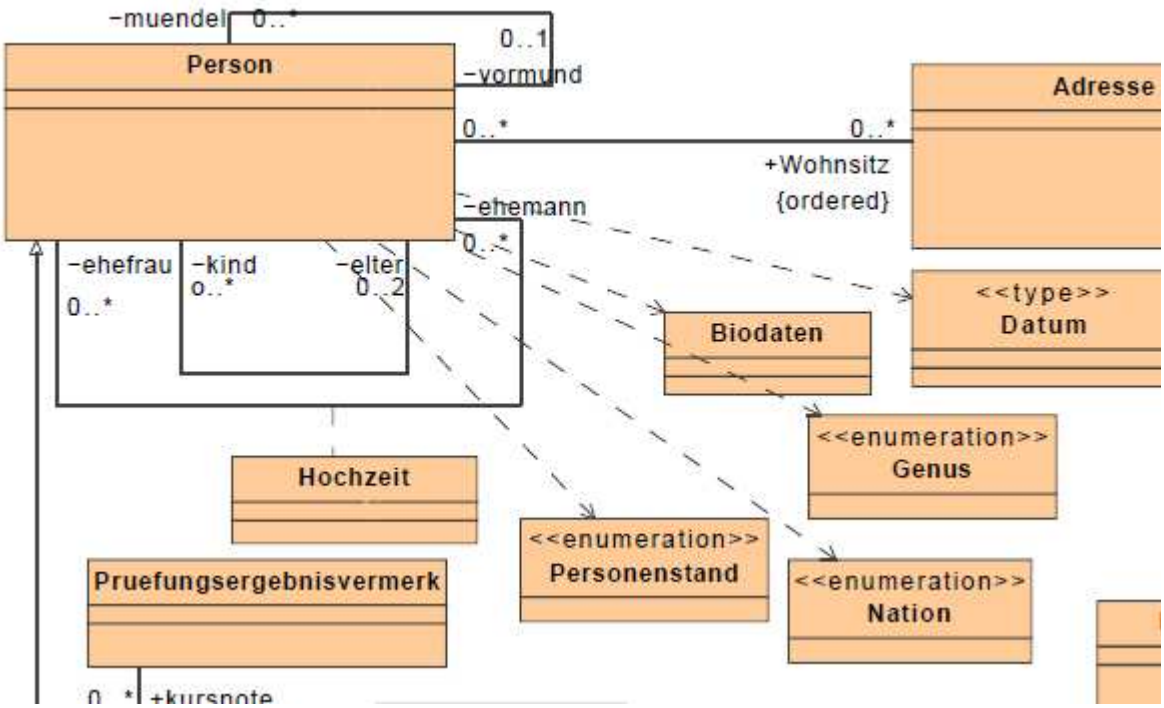
```
    self)
inv : ehefrau->notEmpty() implies ehefrau.ehemann->includes(
    self)
inv : Geschlecht=Genus::maennlich implies ehemann->isEmpty()
inv : Geschlecht=Genus::weiblich  implies ehefrau->isEmpty()
inv : elter->size() = 0 implies not Vormund.oclIsUndefined()

context Person
def: MaennlicheVornamen : Set(String) =
     Set{'Hans', 'Georg' /* bitte ergaenzen */}
def: WeiblicheVornamen : Set(String) =
     Set{'Heidrun', 'Cornelia' /* bitte ergaenzen */}
context Person
def: istMaennlicherVorname(s : String) : Boolean =
     MaennlicheVornamen->includes(s)
def: istWeiblicherVorname(s : String) : Boolean =
     WeiblicheVornamen->includes(s)

context Person
inv : elter->size() > 0 implies elter->forAll( e | (
   Geburtsdatum - e.Geburtsdatum) >= 8)
inv : elter->size() = 1 implies elter->forAll( e | Nachname = e
   .Nachname)
inv : elter->size() = 2 implies
        elter->forAll( e | if e.Geschlecht = Genus::maennlich
           then
                                    e.Hochzeit[ehemann]->size() >
                                        0 implies
                                    e.Hochzeit[ehemann]->forAll(
                                       h |
                                           Geburtsdatum.inbetween(
                                              h.Hochzeitsdatum, h.
                                              Scheidungsdatum)
                                           implies Nachname = h.
                                              Familienname
                                    )
                                else
                                    e.Hochzeit[ehefrau]->size() >
                                        0 implies
                                    e.Hochzeit[ehefrau]->forAll(
                                       h |
                                           Geburtsdatum.inbetween(
                                              h.Hochzeitsdatum, h.
                                              Scheidungsdatum)
```

```
                                        implies Nachname = h.
                                            Familienname
                                    )

                                endif
                            )

context Person
def: keineZeitweiseBigamie( s : Set(Hochzeit) ) : Boolean =
    s->forAll( h1, h2 | h1 <> h2 implies
              (/* h1.Scheidungsdatum >= */ h1.Hochzeitsdatum >=
                 h2.Scheidungsdatum /* >= h2.Hochzeitsdatum */
                 or
               /* h2 Scheidungsdatum >= */ h2.Hochzeitsdatum >=
                 h1.Scheidungsdatum /* <= h1.Hochzeitsdatum */
                 )
              )

context Person
inv : Geschlecht = Genus::maennlich implies
   istMaennlicherVorname(Vorname)
inv : Geschlecht = Genus::maennlich implies Hochzeit[ehemann]->
   size() > 0  implies keineZeitweiseBigamie(Hochzeit[ehemann])
inv : Geschlecht = Genus::weiblich implies istWeiblicherVorname
   (Vorname)
inv : Geschlecht = Genus::weiblich implies Hochzeit[ehefrau]->
   size() > 0 implies keineZeitweiseBigamie(Hochzeit[ehefrau])
inv : Status = Personenstand::tot implies Sterbedatum <> Datum
   ::invalidDatum
inv : Status <> Personenstand::tot implies Sterbedatum = Datum
   ::invalidDatum

context Person::Alter( amDatum : Datum ) : Integer
pre : amDatum >= Geburtsdatum
post : result = amDatum - Geburtsdatum

context Person::Status
init: Personenstand::ledig

context Person::Staatsbuergerschaft
init: Nation::de

context Person::Ausweisnummer
init: 0
```

```
context Person
inv : Geschlecht=Genus::maennlich implies
            Hochzeit[ehemann]->select(Scheidungsdatum=Datum::
               invalidDatum)->size() <= 1

context Person
inv : Geschlecht=Genus::weiblich implies
            Hochzeit[ehefrau]->select(Scheidungsdatum=Datum::
               invalidDatum)->size() <= 1


context Person::Nachname
init: if elter->size() = 1 then
        elter->any(true).Nachname
      else if elter->size() = 2 then
        let aktuelleEheDerMutter : Hochzeit =
           elter->any(Geschlecht=Genus::weiblich).Hochzeit[
              ehefrau]->select(Scheidungsdatum=Datum::
              invalidDatum)->any(true) in
        let aktuellerEhemannDerMutter : Person =
           aktuelleEheDerMutter.ehemann in
        if elter->includes(aktuellerEhemannDerMutter) then
           aktuelleEheDerMutter.Familienname
        else
           elter->any(Geschlecht=Genus::weiblich).Nachname
        endif
      else -- keine Eltern
           'NameVonGericht'
      endif
      endif

/* Verwandschaftsbeziehungen: - allgemeine implizite
   Voraussetzungen: Existenz der jeweiligen Verwandten: */

context Person
def : istNeffe( n : Person ): Boolean =
   elter.kind->asSet()->excluding(self).kind->includes(n) and n
      .Geschlecht=Genus::maennlich

def : Cousins(): Set(Person) =
   (elter.elter.kind->asSet() - elter).kind->asSet()->select(
      Geschlecht=Genus::maennlich)
```

```
def : istStiefvater( s : Person ) : Boolean =
  let mutter : Person = elter->any(Geschlecht=Genus::weiblich)
    in
   (s.Geschlecht=Genus::maennlich) and
   (mutter.ehemann->asSet()-(elter->select(Geschlecht=Genus::
      maennlich)))->includes(s) and
   (s.Hochzeit[ehemann]->select(ehefrau=mutter)->select(
      Scheidungsdatum >= Datum::today())->size() > 0)

context Person::sammleVorfahren( level : Integer ) : Sequence(
   Person)
pre : level >= 0
post :
  if (level = 0) or (elter->size() = 0) then
     result = Sequence{}
  else if elter->size() = 1 then
     result = elter->asSequence()->union(elter->asSequence()->
        at(1).sammleVorfahren( level-1 ))
  else -- elter->size() = 2
   result = elter->asSequence()->union(elter->asSequence()->
                                       at(1).sammleVorfahren(
                                          level-1 ))->
                                    union(elter->asSequence()->at
                                       (2).
                                    sammleVorfahren( level-1 ))
   endif
   endif


endpackage
```

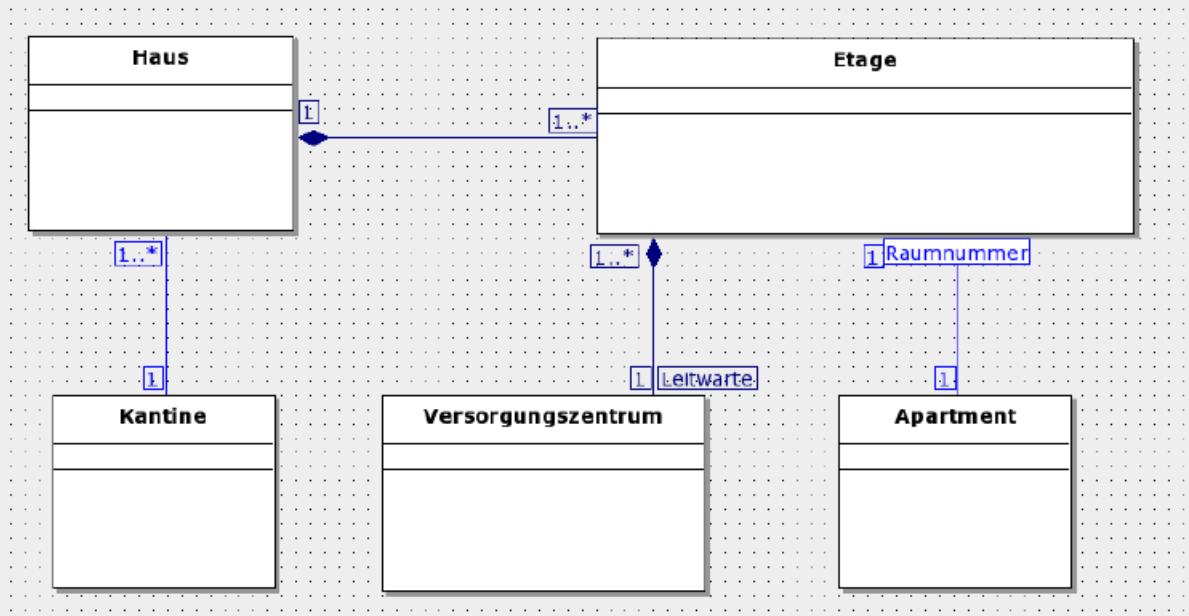
Weitere Assoziationen von `Person` zu `Person`:



Listing 2.12: Constraints Hochzeit

```
package core

context Hochzeit
inv : Familienname <> ''
inv : Scheidungsdatum >= Hochzeitsdatum
inv : Hochzeitsort <> ''
inv : Person[ehemann].Alter(Hochzeitsdatum) >= 14
inv : Person[ehefrau].Alter(Hochzeitsdatum) >= 14
inv : Person[ehemann].Geschlecht = Genus::maennlich
inv : Person[ehefrau].Geschlecht = Genus::weiblich

context Hochzeit::Familienname
init: Person[ehemann].Nachname

context Hochzeit::Scheidungsdatum
init: Datum::invalidDatum

endpackage
```

2.8.16 Modell Wohnanlage



Im Modell `Wohnanlage` werden mehrere Apartmenthäuser, Versorgungszentren (Leitwarten) und Kantinen modelliert. Die Assoziation `etage` sei {`ordered`}.
Jedes mit dem Konstruktor `Haus()` erzeugte Exemplar muß gültig sein, also die Invarianten (und hier insbesondere die konzipierten Vielfachheiten des UML-Modeslls) richtig aufbauen:

Listing 2.13: Constraints Haus()

```
context Haus(in k : Kantine,
             in s : Sequence(Etage))   : Haus
pre:  not k.oclIsUndefined() and
      s->size() > 0
post: result.oclIsNew() and
      result.kantine = k and
      result.etage = s
```

ist deshalg der einzige sinnvolle Konstruktor für die Klasse `Haus`. Will man auch den parameterlosen Default-Konstruktor zulassen, sind die Vielfachheiten im UML-Diagramm anders zu spezifizieren (wie?). Welche Nachteile würde aber eine solche Modellierungsalternative mit sich bringen?

addEtage():

- Nach dem Hinzufügen einer Etage zu einem Haus mittels `Haus::addEtage(e : Etage)` enthält dieses Haus eine Etage mehr als vorher.

  Listing 2.14: Constraint addEtage()

```
context Haus::addEtage(e : Etage)
post: etage->size() - etage@pre->size() = 1
```

- Alle vor Anwendung von `addEtage()` existierenden Etagen sind auch danach noch vorhanden.

```
context Haus::addEtage(e : Etage)
post: self.etage->includesAll(etage@pre)
```

- Eine bereits einem Haus zugeordnete Etage darf keinem (anderen) Haus mehr zugeordnet werden.

```
context Haus::addEtage(e : Etage)
pre: Haus.allInstances().etage->excludes(e)
```

Klassen-Invarianten:

- Kein Haus darf mehr als 10 Etagen besitzen.

```
context Haus
inv: etage->size() <= 10
```

- Jede Etage hat 0..20 Apartments.

```
context Etage
inv: apartment->size() <= 20
```

- Jedes Apartment hat 0..4 Bewohner.

```
context Apartment
inv: bewohner->size() <= 4
                    -- oder
inv: 0 <= AnzBewohner and AnzBewohner <= 4
```

- Jede Kantine hat ein redundantes Attribut `AnzahlHaeuser`, das die Anzahl der assoziierten Häuser beinhaltet.

```
context Kantine::AnzahlHaeuser
derive: haus->size()
```

- Jede Kantine kann höchstens 6 Häuser bedienen.

```
context Kantine
inv: AnzahlHaeuser <= 6
```

Systeminvarianten:

- Jedes Appartment besitzt eine Identifikationsnummer `AppartmentID`, die in der gesamten Wohnanlage eindeutig ist.

```
context Apartment
inv: Apartment.allInstances()->isUnique(AppartmentID)
```

- Jedes Apartment enthält als redundantes Attribut die zugehörige Leitwarte (schnellerer Zugriff bei technischen Problemen).

```
context Apartment::zugLeitwarte
derive: self.etage.leitwarte
```

- Mehrere (verschiedene) Häuser können derselben Kantine zugeordnet sein.

```
context Kantine
inv: haus->size() >= 1
```

- Die Apartments mit Raumnummern kleiner als 20 können höchstens 2 Bewohner aufnehmen.

```
context Apartment
inv: Raumnummer < 20 implies bewohner->size() <= 2
```

- Die Anzahl der Bewohner aller einer Kantine zugeordneten Apartments darf 1000 nicht überschreiten.

```
context Kantine
inv: haus.etage.aparment->collect(AnzBewohner)->sum()<=1000
```

Destruktor-Spezifikation:

- Der Destruktor `ReisseHausAb()` der Klasse `Haus` darf nur aufgerufen werden, wenn alle zugeordneten Apartments keine Bewohner mehr haben.

```
context Haus::ReisseHausAb()
pre: etage->appartment->forAll(AnzBewohner = 0)
```

- Nach Aufruf von `ReisseHausAb()` existieren die zugehörigen Etagen und Apartments nicht mehr.

```
context Haus::ReisseHausAb()
post: Etage.allInstances()->excludesAll(etage@pre)
post: Apartment.allInstances()->excludesAll(etage@pre.
    apartment@pre)
```

- Nach Aufruf von `ReisseHausAb()` existieren alle diejenigen zugehörigen Versorgungszentren nicht mehr, die lediglich für Etagen des abgerissenen Hauses zuständig waren.

```
context Haus::ReisseHausAb()
post: let allVZs : Set(Versorgungszentrum) =
                              etage@pre.leitwarte@pre->asSet()
                                 in
      allVZs->forAll(vz | vz.etage@pre.haus@pre->size() = 1
            implies Versorgungszentrum.allInstances()->
               excludes(vz))
```

2.8.17 Fortsetzung Fallstudie Person/Haus/Hypothek

context Verpfaendung
inv: wert <= hypothek.kreditSumme
inv: wert <= sicherheit.wert
inv: wert >= Euro(0)

context Haus
inv valid_security: hypothek → size()> 0
implies wert >= **v**erpfaendung.wert → sum()

context Hypothek
inv valid_security: **v**erpfaendung → sum() = kreditSumme

context Person
inv: Person.allInstances() → isUnique(ausweisNr)
def: anzHypotheken : Integer = hypotheken → size()

context Person:: nimmHypothekAuf(sum:Euro, sec:Haus)
pre: self.hypotheken.monatlZahlung → sum()+ sum∗0.01 <= self.gehalt∗0.30
pre: sec.wert − sec.verpfaendung.wert→ sum() >= sum
post: ...

context Haus
.... $\underbrace{\text{hypothek} \rightarrow \text{select}(\text{monatlZahlung} > \text{Euro}(500))}$
Menge der Hypotheken auf Haus für die die monatlich Zahlung>500 Euro ist

2.8.18 Startwerte und Ergebnisse von Objekten

context Hypothek::KreditSumme
init: Euro(0)

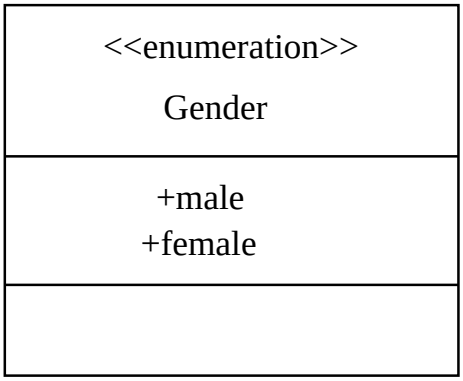
context Hypothek::text
init: schuldner.name.concat(':').concat('- - - - -')

context Haus::getHypotheken(): Set(Hypotheken)
body: hypothek

2.8.19 Virtuelle OCL Variablen / Operationen

context Person
def: income :Integer = self.job.salery → sum()
def: nickname :String = 'little Joe'
def: hasTitle (t:String):Boolean = self.job → exists(title=t)
(Diese OCL-Hilfsobjekte sind im Kontext Person überall benutzbar.)

2.8.20 Enumeration



Benutzung von UML Enumerations

context Person
inv gueltigePersonen : sex = **Gener::male** implies
husband → size()=0

2.8.21 Tuple (records)

... als Ergänzung zur **Sequence/Collection**

Benannte Komponenten bieten Tupel:

(1) Tuple {name:String = 'Hans', age:Integer = 20}

(2) Tuple {a: Collection(Integer) = Set{1,3,4},
b: String = 'foo',
c: String = 'bar'}
(Tuple-Typangaben sind optional und die Reihenfolge der Tuple-Komponenten ist nicht signifikant!)

(3) Tuple {age = 20, name = 'Hans'}

(3) ist also identisch zu (1)

Neben **Tuple-Literalen** gibt es auch Tuple-Typen:
TupleType(name:Strg, age:Int)

2.8.22 Typ-Konformität

OCL-Type

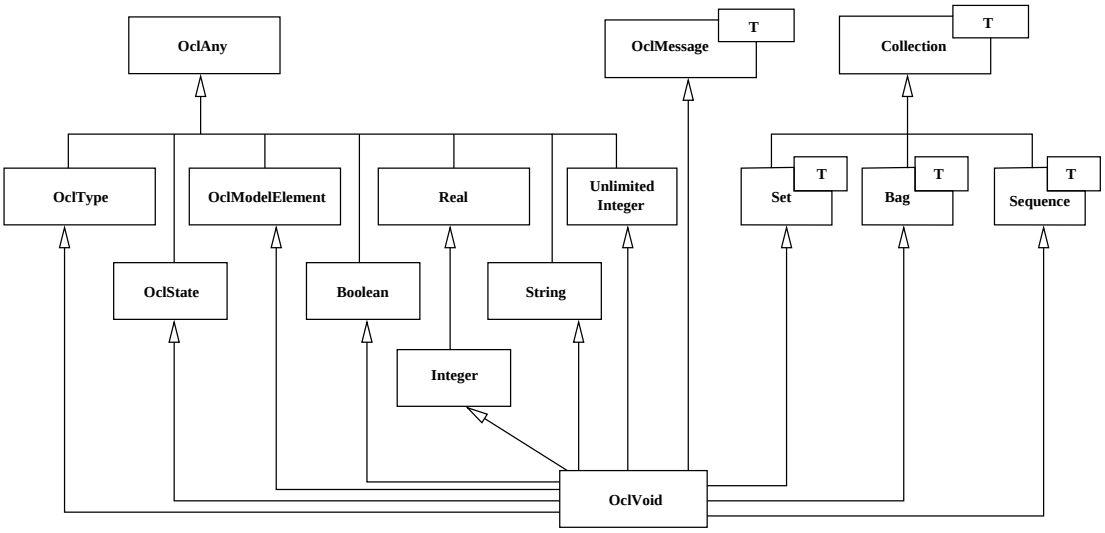


Abbildung 2.12: Die Typen der OCL-Standard-Bibliothek

- Jeder Typ ist konform zu seinen Elter-Typen.
- Die Typenkonformität ist transitiv:

Beispiele:
Integer ist konform zu Real
oclVoid ist konform zu oclAny
Set(T1) ist konform zu Collection(T1)
Set(Integer) ist konform zu Collection(Real)

Sei **obj1** vom Typ OCLAny und es enthalte einen Integer-Wert. Dann erzwingt

obj1.oclAsType(Integer)

die Nutzung von obj1 als Integer.
oclAsType() kann nur in Subtypen wandeln.

2.8.23 Vorrangsregeln

höchste	()
	@pre
	. →
⋮	not - (unär)
	* /
↑	+ - (binär)
	if - then - else - endif
⋮	< > <= >=
	= <>
	and or xor
niedrigste	implies

2.8.24 oclIsUndefined()

Auf Objekten des Typs OclAny kann mittels

obj.oclIsUndefined()

die Existenz abgefragt werden.

2.8.25 Vordefinierte Operationen auf OclType

OclType = (object : OclType) : Boolean
True, falls *self* is dasselbe Objekt wie *object* ist.

OclType <> (object : OclType) : Boolean
post: result = not (self = object)

Weitere Operationen auf **OclType**:

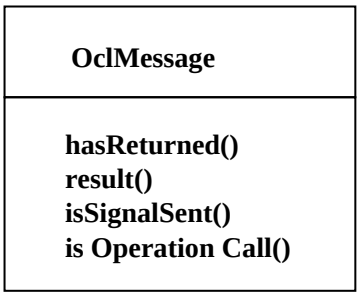
oclIsTypeOf	(t : OclType)	:Boolean
oclIsKindOf	(t : OclType)	:Boolean
oclIsNew	()	:Boolean

2.8.26 Statusdiagramme in UML

Hinweis:
Strukturdiagramme und wechselnde Stati werden in OCL unterstützt durch (siehe Literatur):

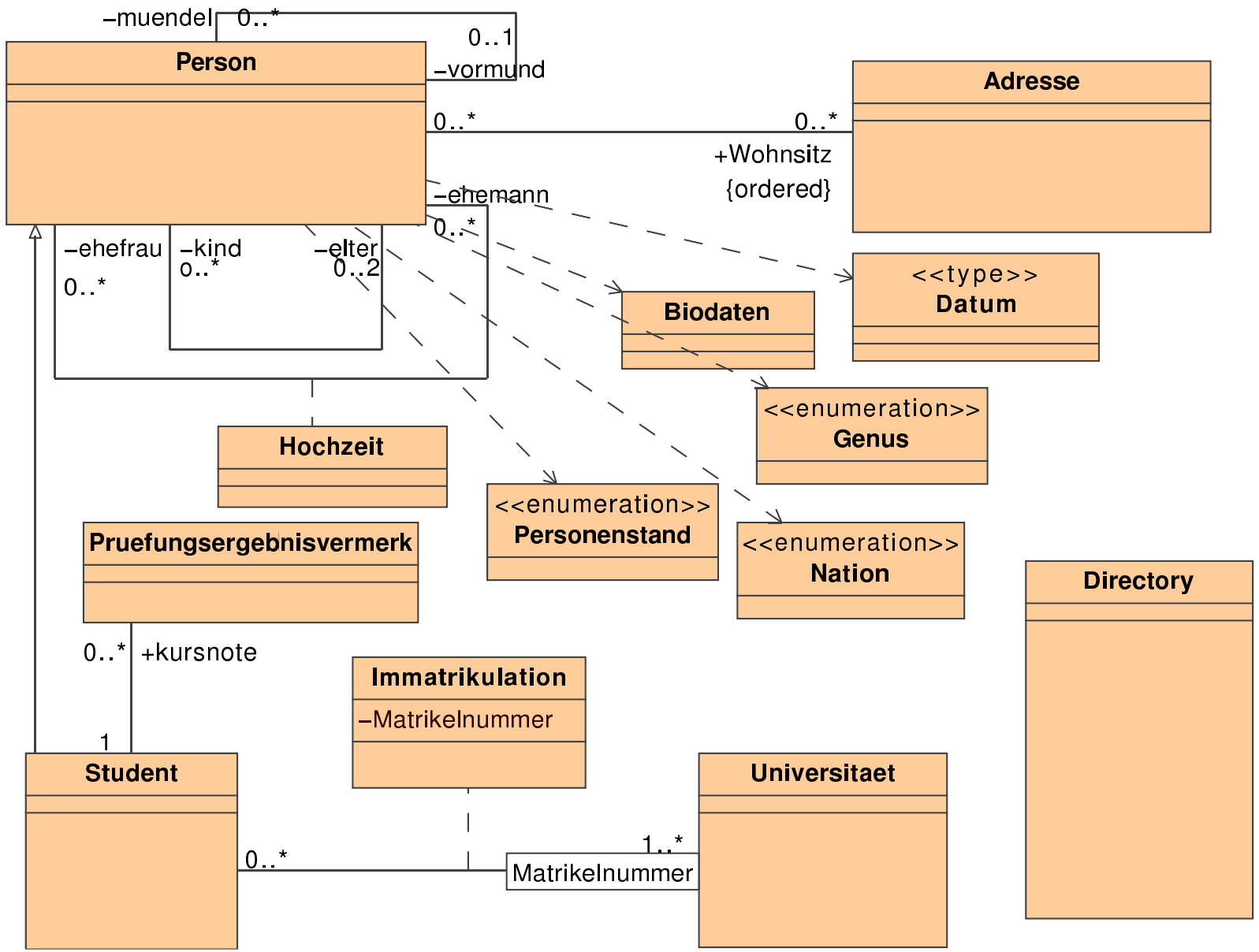
state
oclInState(xx)
OclMessage

Bemerkung:
OclMessage besitzt folgende Methoden:

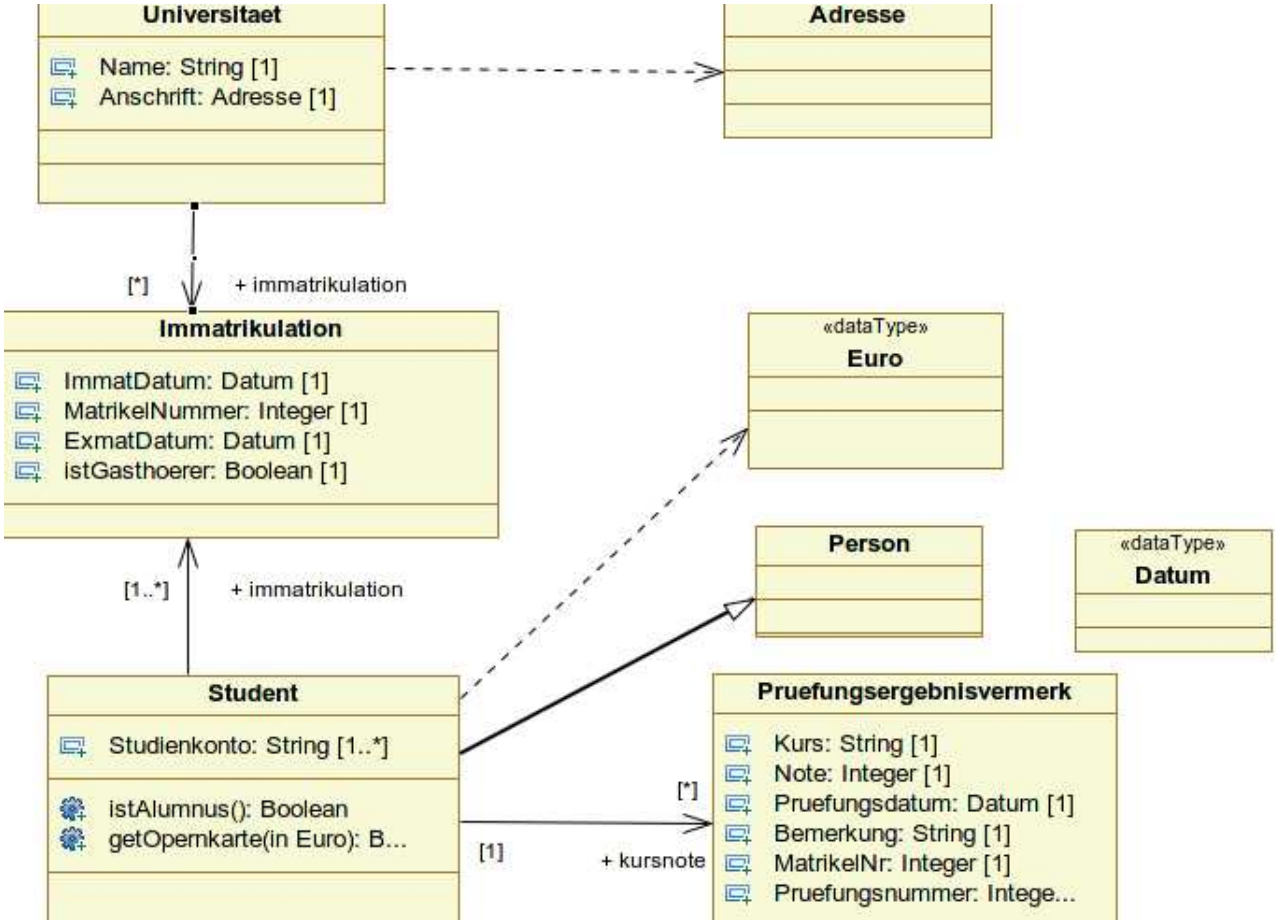


2.8.27 Modell Student/Universitaet/Pruefungsergebnisvermerk

Das UML-Modell mit Assoziationsklassen und qualifizierten Assoziationen:


–muendel
0..*
0..1
Person
–vormund
Adresse
0..*
0..*
+Wohnsitz
{ordered}
–ehemann
0..*
–ehefrau
0..*
–kind
0..*
–elter
0..2
Biodaten
<<type>>
Datum
<<enumeration>>
Genus
Hochzeit
<<enumeration>>
Personenstand
<<enumeration>>
Nation
Pruefungsergebnisvermerk
Directory
0..*
+kursnote
Immatrikulation
–Matrikelnummer
1
Student
Universitaet
0..*
1..*
Matrikelnummer

und seine Pypyrus-Form:



2.8.28 pre-Zustand in Nachbedingungen

7.5.14 Previous Values in Postconditions

As stated in Section 7.3.4, "Pre- and Postconditions," on page 8, OCL can be used to specify pre- and postconditions on operations and methods in UML. In a postcondition, the expression can refer to values for each property of an object at two moments in time:

- the value of a property at the start of the operation or method
- the value of a property upon completion of the operation or method

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword '*@pre*':

context Person::birthdayHappens()
post: age = age@pre + 1

The property *age* refers to the property of the instance of Person that executes the operation. The property *age@pre* refers to the value of the property *age* of the Person that executes the operation, at the start of the operation.

If the property has parameters, the '@pre' is postfixed to the propertyname, before the parameters.

---

context Company::hireEmployee(p : Person)
post: employees = employees@pre->including(p) **and**
stockprice() = stockprice@pre() + 10

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c        -- takes the old value of property b of a, say x
                    -- and then the new value of c of x.
a.b@pre.c@pre    -- takes the old value of property b of a, say x
                    -- and then the old value of c of x.
```

The '@pre' postfix is allowed only in OCL expressions that are part of a Postcondition. Asking for a current property of an object that has been destroyed during execution of the operation results in OclUndefined. Also, referring to the previous value of an object that has been created during execution of the operation results in OclUndefined.

Auch

```
self@pre
```

ist möglich, wie Abschnitt 11.2.5 des OCL-Manuals zeigt.

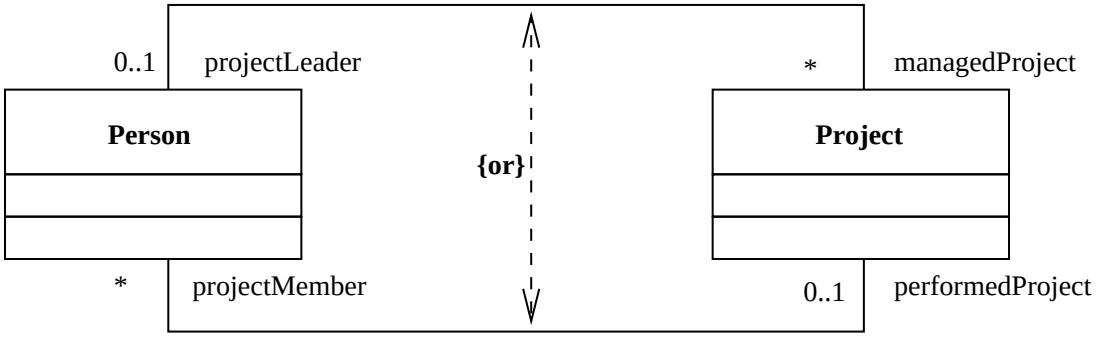
2.8.29 Contracts zum Modell Student/Universitaet/Pruefungsergebnisvermerk

```
package core
context Universitaet
inv : Name <> ''
inv : student->size() >= 0
inv : immatrikulation->isUnique(MatrikelNummer)
inv : student.universitaet->includes(self)
/* nach Einfuehrung einer qualifizierten Assoziation mit
   Assoziationsklasse auch moeglich:
   inv : student[0312345].Familienname = 'Bauer'
*/
endpackage -- core

package core
context Student
inv : immatrikulation->size() >= 1
inv : immatrikulation->select(not istGasthoerer)->size() >= 1
inv : universitaet.student->includes(self)
inv : pruefungsergebnisvermerk->size() > 0 implies
        pruefungsergebnisvermerk.student->asSet() = Set{self}

context Student::istAlumnus() : Boolean
body : not immatrikulation->exists(ExmatDatum.oclIsUndefined())

context Student::getOpernKarte( Einzahlung : Euro ) : Boolean
pre : Einzahlung >= Euro(10)-- allg. Vorbedingung als in Person
post : True
endpackage -- core

package core
context Immatrikulation
inv : MatrikelNummer > 0
inv : ExmatDatum <> OclUndefined implies ExmatDatum >=
   ImmatDatum
/* oder: inv : ExmatDatum.oclIsUndefined() implies ExmatDatum
   >= ImmatDatum */

inv : ImmatDatum - student.Geburtsdatum >= 12
inv : ExmatDatum <> OclUndefined implies Datum::today() >=
   ExmatDatum

context Immatrikulation::ImmatDatum
```

```
init : Datum::today()

context Immatrikulation::ExmatDatum
init : OclUndefined -- 2. Versuch eines optionalen Datentyps:
                    -- hier: OclUndefined als Ungueltig-Marke

context Immatrikulation::istGasthoerer
init : false
endpackage

package core
context Pruefungsergebnisvermerk
inv : Kurs <> ''
inv : 1 <= Note and Note <= 5
inv : MatrikelNr > 0
inv : student.immatrikulation.MatrikelNummer->includes(
   MatrikelNr)
inv : Pruefungsdatum.inbetween(student.immatrikulation->any(
   MatrikelNummer=MatrikelNr).ImmatDatum,
        student.immatrikulation->any(MatrikelNummer=MatrikelNr).
           ExmatDatum)
inv : Pruefungsnummer > 0
inv : Pruefungsergebnisvermerk->allInstances()->isUnique(
   Pruefungsnummer)
inv : Student.kursnote->includes(self)
-- spaeter besser: inv : student.kursnote[Pruefungsnummer] =
   self

-- wo steckt der Modell-Fehler?
endpackage -- core
```

2.9 UML Constraints

2.9.1 or / xor



ist **mehrdeutig:**

Bezieht sich das **or** auf die linke oder die rechte Assoziationsendseite?

OCL kann die betreffende Seite eindeutig beschreiben:

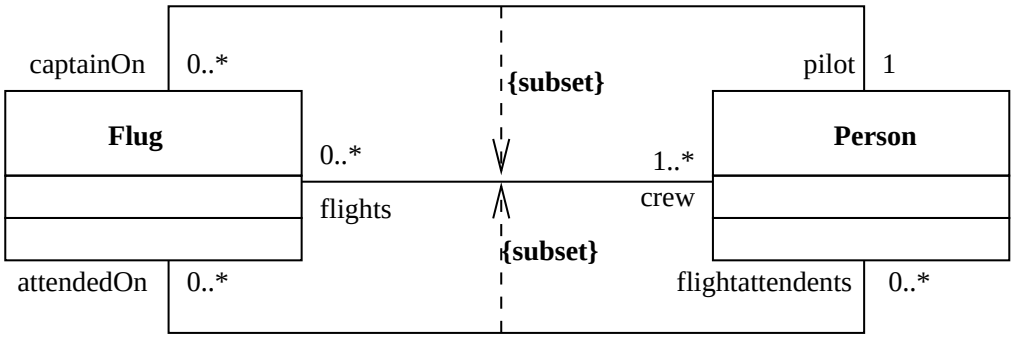
context Person
inv: not managedProject → isEmpty() or
not performedProject → isEmpty()

Jede Person leitet entweder das Projekt oder arbeitet an ihm mit.

(oder falls die linken Assoziationsenden von Bedeutung:)
context Project
inv: not projectLeader → isEmpty() or
not projectMember → isEmpty()

Jedes Projekt hat entweder ein Projektleiter oder ein Projektmitglied.

2.9.2 subset



Subset-Constructs machen UML-Diagramme häufig schlecht lesbar.
Besser ist:

context Flug
inv: self.crew → includes(self.pilot)
inv: self.crew → includesAll(self.flightattendents)

oder:

context Person
inv: self.flights → includesAll(self.captainOn)
inv: self.flights → includesAll(self.attendedOn)

2.10 Stil-Hinweise für OCL-Constraints

- Schreibe lieber für viele Klassen kurze OCL-Ausdrücke, die nur wenige Assoziationen tief „navigieren“, als lange Navigationsketten, die das ganze Modell durchlaufen.

- Vermeide allInstances() wenn immer möglich:
  Zum Beispiel ist

  context Person

  inv: parents → size() <= 2

  und

  context Person

  inv: Person.allInstances() → forAll(p | p.parents → size() <= 2)

  äquivalent, aber unterschiedlich effizient.

- Nutze Invarianten in Klassen, um die möglichen Werte der Attribute einer Klasse von den unmöglichen zu unterscheiden.

- Schreibe Invarianten in die Klasse, zu der sie gehören:
  - Attributwerteinschränkungen gehören in die Klasse, die das Attribut definieren.

– Falls eine Invariante die Attribute mehr als einer Klasse einschränkt, kann jede dieser Klassen als Kontext gewählt werden. Eventuell kann man einer dieser Klassen die Verantwortung über die andere zuteilen.

– Jede Invariante sollte durch möglichst wenig Assoziation navigieren.

– Versuche bei Bedarf, eine Invariante testweise im Kontext verschiedener Klassen zu formulieren. Wähle dann die einfachste Version.
Zum Beispiel ist

```
Context Company
inv: employees.wife -> intersection(
        self.employees) -> isEmpty()
```

einfacher als

```
Context Person
inv: wife -> notEmpty() implies
        wife.employers -> intersection(self.employers)
                -> isEmpty()
```

– Nutze viele einfache,

```
inv: ...
inv: ...
```

statt einer einzelnen inv-Zeile komplizierterer Bauart.

- Vermeide der Lesbarkeit halber collect():

```
context Person
inv: self.parents.brothers.children → notEmpty()
```

ist äquivalent zu:

```
context Person
inv: self.parents →
        collect(brothers) →
        collect(children) → notEmpty()
```

- Gib allen Assoziationsenden einen Namen.

2.10.1 Einfache Beispielverträge und die geeignete Kontextwahl

Siehe: Introduction to OCL

2.11 OCL in Together-Produkten

mit OCL-Constraints, Code-Erzeugung für diese, ...

OCL Basic Types
OCL Collection Types
Tic-Tac-Toe Example
Gererating Code from OCL

2.12 OCL 2.2 / May 2009 — Die Änderungen

Link

- Stings, Literale, Typnamen, Feature-Namen, ... in UNICODE
- static def's, `Classname::Id`
- in Konflikt stehende Rollennamen durch nichtleere Assoziationsnamen benutzbar machen
- `OCLType` durch `Classifier` für alle Objekte ersetzt
- `Classname.AllInstances()`
- `feature` statt `property`
- `behaviour` statt `method`
- `@`pre ist Postfix zum Operationsnamen, nicht zum Operationsaufruf
- Zugriff auf ein nicht mehr existierendes Feature erzeugt `null` statt `OclUndefined`
- `exists` darf analog zu `forAll` mehrere Iteratoren (Laufindices) definieren
- `TypeType` und `ElementType` werden aus OCL entfernt
- *CollectionType* wird zu `CollectionType` (konkrete statt abstrakter Elterklasse)
- neu: `TemplateParametertype`
- Collectionkind enthält jetzt auch `Collection`
- **context** Classifier
  def: allInstances() : **Set**(T)
- Operator-Prioritäten: "`^`" und "`^^`" (messages-expression) hat (neu) zweithöchste Priorität

- Operator-Prioritäten: `let` ist in die Prioritäten-Liste mit geringster Priorität aufgenommen worden
- `OclAny` ist Eltertyp aller Typen (der selbsdefinierten, der primitive UML-Typen, der Collection-Typen)
- `OclVoid` kann allen Typen außer `OclInvalid` zugewiesen werden. Es hat eine einzige Instanz (`null` = UML Literal-Null). Feature-Calls von `null` resultieren in `OclInvalid`. Lediglich `oclIsInvalid`, `oclIsUndefined` und `isEmpty` sind erlaubt. Auf Collection-Operationen wird `null` als `Bag{}` interpretiert.
- Redefinition von

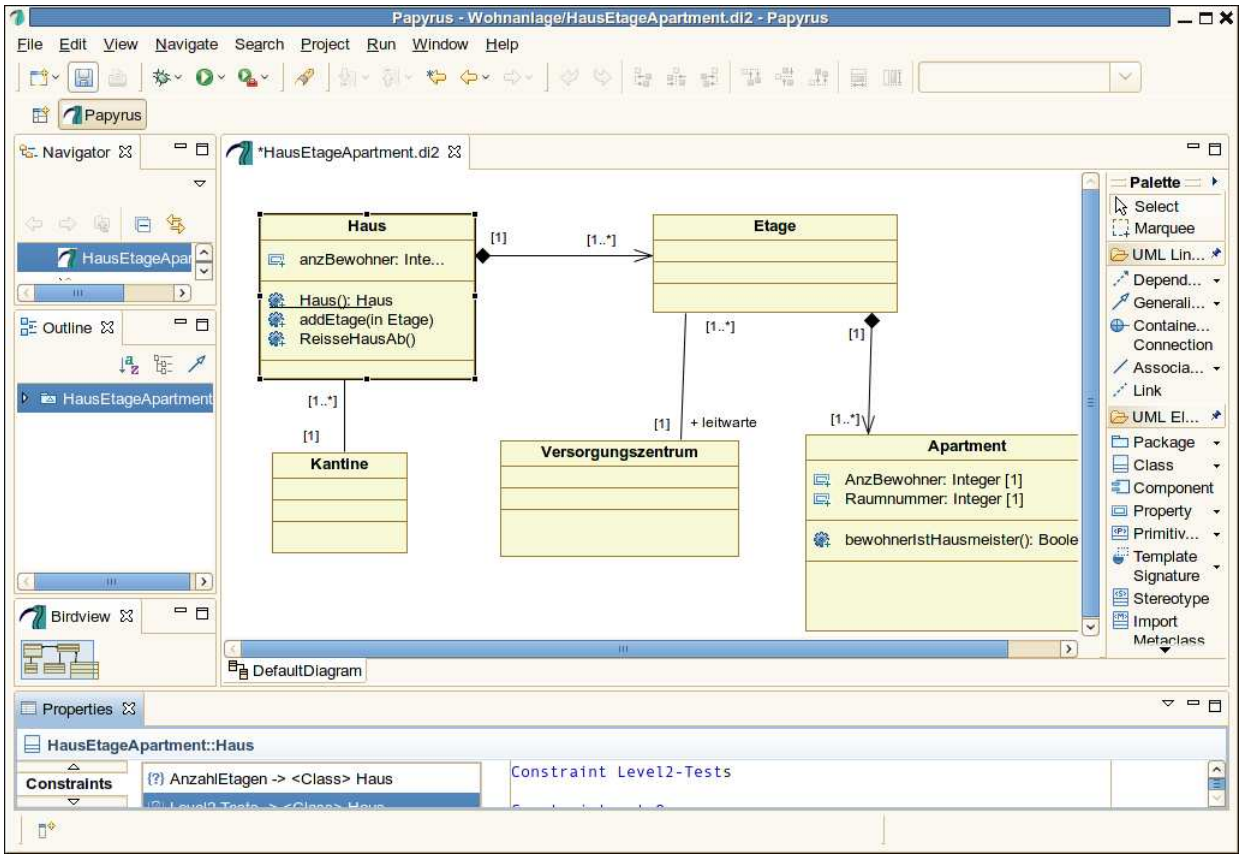
  OclVoid = (obj : **OclAny**) : **Boolean**
  post: result = obj.oclIsTypeOf(OclVoid)

  OclInvalid = (obj : **OclAny**) : **Boolean**
  post: result = obj.oclIsTypeOf(OclInvalid)

- `UnlimitedInteger` wird durch `UnlimitedNatural` ersetzt
- `toString(): String` wird für `Integer, Real, Boolean` eingeführt
- `String +(s : String) : STRING`
- neue String-Operationen: `toUpperCase(): String, toLowerCase(): String, indexOf(s: String): Integer, equalsIgnoreCase(s : String): Boolean, at(i : Integer): String, characters(): Sequence(String), toBoolean(): Boolean)`
- `OrderedSet` ist weder ein `Set` noch eine `Sequence` sondern eine `Collection`
- `Collection =(c: Collection): Booolean`
  True iff c und self von derselben Art sind und dieselben Elemente in derselben Anzahl (in derselben Reihenfolge, falls `ordered`) enthaltend
- `null->isEmpty()` ist `True`
- `Collection max(): T`, sofern für Komponenten existent und assoziativ und kommutativ
- `Collection min(): T`, sofern für Komponenten existent und assoziativ und kommutativ
- `Collection asSet(): Set(T), asOrderedSet(): OrderedSet(T), ...`
- `Collection flatten(): Collection(T2)`
- `Collection`s können keine `OclInvalid`-Werte enthalten

- `OrderedSet reverse(): OrderedSet(T)`
- `Sequence reverse(): Sequence(T)`
- `collectNested()` verbesserte Erläuterung
- `OrderedSet select(exp: OclExpression): OrderedSet(T),`
  `OrderedSet reject(exp: OclExpression): OrderedSet(T),`
  `OrderedSet collectNested(exp: OclExpression): Sequence(T),`
  `OrderedSet sortedBy(exp: OclExpression): OrderedSet(T)`

2.13 Metalevel2-Constraints = Wohldefiniertheitsregeln für Modelle

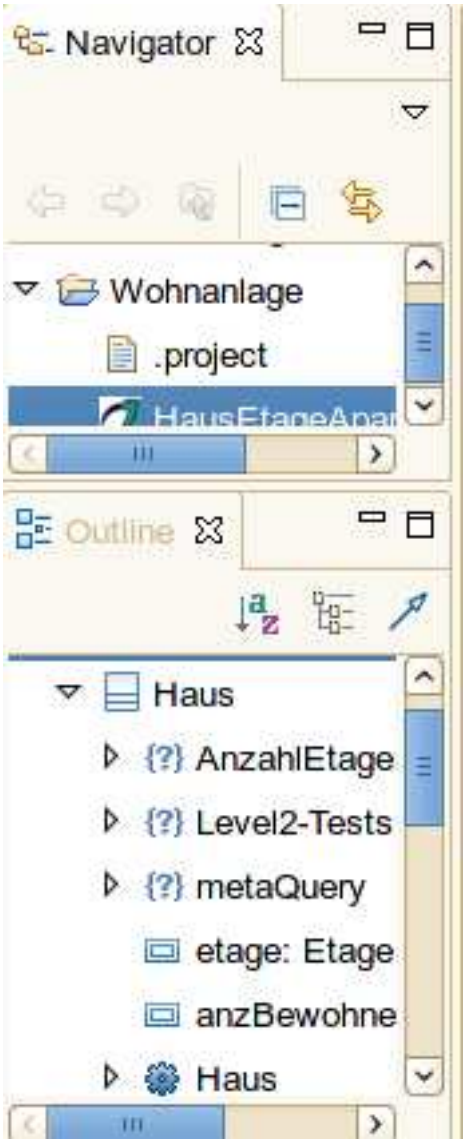


Im M2-Level des Constraints-Checkers von Papyrus kann man ein Modell algorithmisch mit OCL-Ausdrücken abfragen (Query-Sprache) und zum Beispiel Regeln des Programmiererteams als Invarianten definieren. Einige Queries:

Context Haus
self — *Class Haus*

```
self.name                                  -- Haus

self.feature->size()                       -- 5
self.feature->asSequence()->at(1)          -- operation ReisseHausAb

namespace.ownedElement->size()             -- 11
```

Die Modellelemente können Sie natürlich auch im Papyrus-Outline-Fenster betrachten:



Einige WFR-Regeln:

```
context Haus
inv: namespace.ownedElement->collect(name)->count(self.name)=1
```

```
context Class
inv: not self.name.oclIsUndefined() and self.name <> ''

context ModelElement
inv: NamedElement.allInstances()->forAll(c1,c2|c1<>c2 implies
    c1.name<>c2.name)

context Model
inv: Class.allInstances()->collect(c:Class| not c.name.
    olcIsUndefined())->size()=1
```

modeling levels: AST, UML-model/SdV, Oject-diagram

```
"The name of an opposite AssociationEnd
may not be the same as the name
of an Attribute or a ModelElement
contained in the Classifier."

context Classifier
inv WFR_5:
self.oppositeAssociationEnds->forAll(o |
not self.allAttributes->union(self.allContents)->collect(
q | q.name)->includes(o.name))


"The name of an Attribute may not be
the same as the name of an opposite
AssociationEnd or a ModelElement
contained in the Classifier."

context Classifier
inv WFR_4:
self.feature->select(a | a.oclIsKindOf (Attribute))->forAll(a |
not self.allOppositeAssociationEnds->union(self.allContents)
->collect(q | q.name)->includes(a.name))
```

M2 Modellierungsrichtlinien, WFRs

M1 Geschäftsregeln, SdV, DbC, Spezifikation von Testfällen

M0 Ausführung von Testfällen

http://wiki.eclipse.org/OCLSnippets

2.14 OCL und die Modell-Transformation im MDA

http://wiki.eclipse.org/ATL/User_Guide

UML-Metamodell

Biblio-Metamodell

2.15 OCL-Beispiele

http://www.empowertec.de/ocl/example1.htm
...